Graphes "State Constraint Transition" : un langage pour la spécification formelle des systèmes de systèmes dynamiques

Asmaa Achtaich

École Mohammadia D'ingenieurs
Université Mohammed V de Rabat

Université Paris 1
Panthéon-Sorbonne

THÈSE EN COTUTELLE

Pour obtenir le grade de : **Docteur en Sciences et Techniques pour l'Ingénieur**
Et le grade de : **Docteur de l'Université Paris 1 Panthéon-Sorbonne**
Spécialité : Informatique

Graphe « State Constraint Transition »
Un langage pour la spécification formelle des systèmes de systèmes dynamiques

---

Présentée et soutenue par :

Asmaa ACHTAICH

Le : 23/10/2020

Devant le jury composé de

Pr. Mohsine Eleuldj *EMI – Université Mohammed V de Rabat - Maroc*	**Président**
Pr. Philippe Collet *Université Côte d'Azur – Nice - Fance*	**Rapporteur**
Pr. Adil Anwar *EMI – Université Mohammed V de Rabat - Maroc*	**Rapporteur**
Pr. Omar El Beqqali *FS Dhar El Mahraz –Université Sidi Mohammed Ben Abdallah – Fès - Maroc*	**Rapporteur**
Pr. Ounsa Roudiès *EMI – Université Mohammed V de Rabat - Maroc*	**Directeur**
Pr. Camille Salinesi *Université Paris 1 Panthéon-Sorbonne – Paris - France*	**Directeur**
Pr. Nissrine Souissi *École Nationale Supérieure des Mines de Rabat - Maroc*	**Co-directeur**
Pr. Raul Mazo *École Nationale Supérieure de Techniques Avancées Bretagne - France*	**Co-directeur**

Remerciements

Durant toutes les années de réalisation de cette thèse, innombrables étaient les fois où je me suis projetée dans le futur, des fois proches, des fois infinies, songeant et sollicitant le jour où je pourrai enfin rédiger cette section. L'aventure de cette thèse, comme la plupart d'entre vous l'ont certainement eux même vécu, est un conte, dont les protagonistes sont nombreux. Et aujourd'hui, j'ai enfin la chance et l'immense honneur de leur adresser ma profonde gratitude et reconnaissance.

Tout d'abord, à ma Directrice de thèse, Madame Ounsa ROUDIES, pour sa patience, son amabilité, son soutien académique et moral quand rien ne va, et pour sa disponibilité, ses encouragements et sa confiance. Votre rigueur, votre respect pour la science, et votre acharnement pour l'excellence sont admirables. J'ai certainement beaucoup appris de vos enseignements et vos conseils. Je suis aussi une femme meilleure, rien qu'en vous côtoyant.

A mon Directeur de thèse, Monsieur Camille SALINESI. Un grand Professeur, un grand Homme. Je tiens à vous exprimer mes sincères respects pour vos valeurs pédagogiques et humaines. Vous avez toujours témoigné d'une maturité, d'une sagesse et d'un savoir inégalés dans tous vos conseils, décisions et recommandations. Je tiens particulièrement à vous remercier d'avoir accepté à diriger cette thèse en cotutelle, elle n'aurait certainement pas abouti de la sorte sans vos directions et vos challenges, et pour cela, je vous suis très reconnaissante.

A ma co-directrice de thèse, Madame Nissrine SOUISSI, sans laquelle, rien de cela ne serait. Je vous remercie pour votre confiance, pour votre disponibilité et réactivité, dans chaque détail, et la moindre des complications. Je vous remercie d'être toujours à l'écoute, attentive, mais surtout, efficace. Votre esprit critique et méthodique a apporté à cette thèse la structure qui lui a souvent manqué. Je vous suis très reconnaissante pour tous vos enseignements ; sérieux, rigueur, et assiduité, avec un soupçon de tolérance et assurance.

A mon co-directeur de thèse, Monsieur Raul MAZO, pour son savoir, sa rigueur et son esprit perfectionniste. Je vous remercie pour la disponibilité dont vous m'avez témoigné dans toutes les activités de recherche ; depuis l'assimilation jusqu'à la rédaction. Je vous remercie pour vos encouragements et vos critiques, pour votre amitié, pour nos échanges, et pour votre grande générosité.

Mes sincères remerciements vont aux membres du jury, pour leur disponibilité, et l'attention qu'ils ont porté à bien vouloir participer à l'examen de ce travail.

A Monsieur Mohsine ELEUDJ, président du jury, je vous remercie d'avoir pris de votre temps et vos ressources pour présider le jury de ma soutenance de thèse. Je suis honorée, et je tiens à vous exprimer mon immense joie et gratitude pour votre disponibilité.

A Monsieur Philippe COLLET, Monsieur Adil ANWAR et Monsieur Omar EL BEQQALI, merci d'avoir accepté de rapporter cette thèse. Merci pour vos remarques pertinentes,

pour vos critiques constructives, et pour le temps que vous avez consacré pour accroitre la qualité de ce mémoire.

Je tiens à remercier tous les membres de la structure de recherche Siweb à l'École Mohammadia d'Ingénieurs, les professeurs Anouar, Assoul, Ouazzani de toujours garder leurs portes ouvertes à mes sollicitations, ainsi que tous les collègues, Farid, Ikram, Anass, Fadwa, Sihame, Mohammed, Younes, Hind, Naima … Merci pour votre amitié, vos conseils, et pour tous nos échanges.

Je témoigne de ma sincère gratitude à tous les membres du Centre de Recherche en Informatique à l'Université Paris 1 Panthéon - Sorbonne, et principalement, au Dr. Diaz, pour ses conseils et son âme attentionnée et généreuse, et au Pr. Le Grand, pour l'exemple qu'elle représente de sérieux, d'attention et d'amabilité. Un grand merci aux Pr Rolland, et Souveyet, et aux Dr Kirsch Pinheiro, Deneckère, Rychkova et Nurcan pour leur accueil, intégration et orientation.

A l'équipe de thésards du CRI, un grand Merci.

A « mama » Angela, merci d'avoir « toujours » été à l'écoute. Merci pour tes directives et conseils. Merci pour ton cœur, et merci pour être ma famille dans mes jours les plus joyeux et les plus lugubres.

A Elena E., merci pour nos longues marches et discussions nocturnes, merci pour ton amitié rafraichissante et unique. Et merci pour l'équilibre que tu apportes à tout ce que tu frôles.

A Luisa, pour ton esprit brillant et généreux, à Danny, pour ses petits mots, et son grand impact, à Elena K., pour sa force et sa douceur, à Ali, pour son esprit de leadership, à Danillo, pour son âme généreuse, à David, pour sa bonne humeur, dans les jours les plus durs et les plus doux, à Amina, pour sa grandeur d'esprit, à Juan Carlos Muñoz, pour sa patiente et sa collaboration, à Raouia, pour son accueil et sympathie, à Juan-Carlos Martinez, pour sa gentillesse inégalée, et à Fabrice, pour ses discussions extraordinaires. Un grand merci à Abir, Sabrine, Houssam, Abdelkader, Afef, Indra, Lookman et Lamiae pour tout leur soutien.

Je remercie, ma famille, pour leur soutien et leur prières. Mes amis (dans l'ordre chronologique), Laila et toute sa famille, Lamia, Zineb, Anas, Barkahoum, Zakarya, Assia, Abir, Asma, Meriem. Merci de m'avoir rassuré. Merci de m'avoir nourri, diverti, supporté. Merci d'être. Ma sœur Khadija, d'avoir toujours veillé sur moi. Ma Didou, d'être une lumière dans mes jours les plus lugubres. Ma sœur Hind, merci d'être prête à aller au bout du monde pour moi, figurativement et littéralement. Ma mère Amina, pour ses prières quand tous dorment, pour sa confiance, pour sa voie et voix immortelles. Mon père Naceur, pour son support et son amour inconditionnel, pour tout ce qu'il dit, sans rien dire.

TJ. Pour tout ce qu'on a vécu. Pour tout ce qu'on sait. Pour tout ce qu'on sent. Pour tout ce qu'on sera.

Noor, et ceux qui m'illumineront après.

Résumé

Les systèmes de systèmes dynamiques sont composés de systèmes intelligents qui collaborent pour offrir des services. Souvent déployés dans des environnements dynamiques et hétérogènes, et grâce aux technologies de pointe de plus en plus accessibles et répandues, les systèmes de systèmes sont capables de collecter et communiquer l'information sur leur contexte en temps réel, notamment en vue d'en faire l'analyse. Un autre avantage important d'être constamment à l'écoute de cet environnement est la possibilité de se reconfigurer, éventuellement à distance, de manière à assurer une qualité et une durabilité de service conforme aux attentes des utilisateurs. Par ailleurs, les systèmes de systèmes sont par définition composés de sous-systèmes hétérogènes et indépendants, qui collaborent pour atteindre un objectif ultime. Ces caractéristiques, couplées à la nature dynamique des exigences font que leur spécification devient complexe. Cette thèse, consacrée à la spécification et la configuration des systèmes de systèmes dynamiques, prend le parti de se baser sur des principes de réutilisation pour la dérivation de configurations en vue de réaliser ceux-ci au fur et à mesure que le contexte ou les exigences évoluent. Le cadre des lignes de produits logiciels dynamiques (LdPD) est ainsi adapté puis exploité en tant que base conceptuelle. Le cas détaillé d'un système de systèmes dynamique d'irrigation « GreenLife » sert à illustrer la problématique et la typologie des exigences exprimées pour ce type de systèmes, et à présenter les démarches de spécification et de configuration proposées.

La contribution principale réside dans la notation formelle State-Constraint Transitions (***SCT***). ***SCT*** est dédié à la spécification de systèmes de systèmes dynamiques, qui adaptent leur comportement en réponse à l'évolution du contexte et des exigences. D'un point de vue conceptuel, ***SCT*** est une variante des machines à états finis dont la puissance d'expression est étendue au moyen du concept de contraintes. Ce langage de spécification apporte une réponse aux verrous liés à la spécification d'exigences dynamiques en introduisant un concept d'état de configuration traduisant les exigences en contraintes qui permettent de contrôler l'adéquation des configurations à des contextes variables. Le pouvoir d'expression des approches LdPD existantes est ainsi étendu, en combinant la facilité d'utilisation des notations bien établies (notamment celles basées sur les caractéristiques, et celles basées sur les machines à états), avec la puissance d'expression et de calcul de la programmation par contraintes. Cette contribution est mise en œuvre au moyen du Framework Xtext. Elle a été évaluée avec trois cas d'application issus de trois domaines complètement différents : le système d'irrigation GreenLife, le réseau de capteurs sans fils Gridstix, et le système de train d'atterrissage LGS. L'évaluation est faite selon deux points de vue. D'un côté, elle montre l'indépendance du langage SCT par rapport à un domaine spécifique, évalue son pouvoir d'expression et sa capacité à passer à l'échelle pour spécifier des systèmes de grande taille. Et de l'autre, elle décrit les propriétés contribuant à l'efficacité perceptive de la notation SCT.

Mots clés : système de systèmes dynamiques, internet des objets, lignes de produits logiciels dynamiques, machines à états finis, contraintes, ingénierie des exigences, langage de spécification, systèmes complexes

Abstract

Dynamic systems of systems (DySoS) are made up of intelligent systems that work together to deliver specific services. Often deployed in dynamic and heterogeneous environments, and thanks to increasingly accessible and widespread advanced technologies, these systems are capable of collecting and communicating information on their context in real time, in order to analyze it, and eventually act according deducted results. Another important advantage of being aware of the environment is the possibility of being self-reconfiguring, possibly remotely, so as to ensure quality and durability of service in line with user expectations. Furthermore, systems of systems are composed of a collection of heterogeneous and independent subsystems, which collaborate to achieve an ultimate objective. These characteristics, coupled with the dynamic nature of requirements, make their specification complex. This thesis, devoted to the specification and configuration of dynamic systems of systems, dawns from reuse principles for the derivation of configurations, which are achieved as the context or requirements evolve. The framework of dynamic software product lines (DSPL) is thus adapted and then used as a conceptual basis. The detailed case of a dynamic irrigation system of systems, "GreenLife", serves to illustrate the problem and the typology of requirements expressed for this type of systems, and to present the proposed specification and configuration approaches. The main contribution lies in the State-Constraint Transitions (SCT) formal language. SCT is dedicated to the specification of dynamic systems of systems, which adapt their behavior in response to changing contexts and requirements. From a conceptual point of view, SCT is a variant of finite state machines (FSM) whose power of expression is extended by means of the concept of constraints. This modeling language provides an answer to the problems linked to the specification of dynamic requirements by introducing the concept of configuration states, in which requirements are translated into constraints. This makes it possible to control the adequacy of the configurations in variable contexts. The expressiveness of existing DSPL approaches is thus extended, combining the ease of use of well-established notations (notably those based on characteristics, and those based on FSM), with the computational power and expressiveness of the constraint programming approach. This contribution is implemented using the Xtext framework and is evaluated with three application cases from three different fields: the GreenLife irrigation system, the Gridstix wireless sensor network, and the LGS landing gear system. The assessment is made from two points of view. On the one hand, it shows the independence of the SCT language in relation to a specific domain, assesses its expressiveness and its ability to scale to specify large systems. And on the other hand, it describes the properties contributing to the perceptive efficiency of the SCT notation.

Keywords : dynamics systems of systems, internet of things, fleet, dynamic software product lines, finite state machines, constraints, requirements engineering, modeling language

Publications

I. Publication dans des revues à comité de lecture indexées dans des bases internationales :

1. Achtaich, A., Salinesi, C., Souissi, N., Mazo, R and Roudies, O. Guidelines for the Specification of IoT Requirements: The case of smart cars. IoT Protocols and Applications for Improving Industry, Environment, and Society. Book Chapter, Book Chapter. (2020). DOI: 10.4018/978-1-7998-6463-9
2. Achtaich, A., Souissi, N., Roudies, O., Salinesi, C. and Mazo, R. A Constraint-based Approach to Deal with Self-Adaptation: The Case of Smart Irrigation Systems. International Journal of Advanced Computer Science and Applications. Vol. 10, No. 7, (2019).
   DOI : 10.14569/IJACSA.2019.0100727
3. Achtaich, A., Mazo, R., Souissi, N., Salinesi, C., Roudies, O. Management Capabilities for Mobile and IoT Devices: An Evaluation Framework. International Journal of Engineering and Advanced Technology (IJEAT)', ISSN: 2249-8958 (Online), Volume-8 Issue-6, Page No.:420-430. (2019).
   DOI : 10.35940/ijeat.E7822.088619
4. Achtaich, A., Souissi, N., Mazo, R., Roudies, O., Salinesi, C. A DSPL Design Framework for SASs: A Smart Building Example. EAI Endorsed Transactions on Smart Cities. Volume 2, Issue 8, e1 (2018).
   DOI : 10.4108/eai.26-6-2018.154829

II. Communication dans des conférences à comité de lecture indexées dans des bases internationales avec publication d'actes :

5. Salinesi, C., Rohleder, C., Achtaich, A., Kusumah, I., Innocrowd, A Contribution To An Iot Based Engineering Product Development, *International Conference on Computer Science and Information Technology (COSIT),* Zurich, Switzerland (2020)
6. Achtaich, A., Roudies, O., Souissi, N., Salinesi, C., Mazo, R. Evaluation of the State-Constraint Transition Modelling Language: A Goal Question Metric Approach.

Proceedings of the 23rd International Systems and Software Product Line Conference (SPLC), Paris, France (2019)
DOI: 10.1145/3307630.3342417

7. Achtaich, A., Souissi, N., Mazo, R., Salinesi, C. , Roudies, O. Designing a Framework for Smart IoT Adaptations. Proceedings of the 1st International EAI Conference on Emerging Technologies for Developing Countries. (AFRICATEK), Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Volume 206, ISBN 978-3-319-67836-8, Marrakech, Morocco (2017).
DOI: 10.1007/978-3-319-67837-5_6
8. Achtaich, A.Vers une plateforme de monitoring de flottes d'objets connectés, *Actes du 8 e Forum Jeunes Chercheurs du congrès INFORSID*, Grenoble, France (2016)
9. Achtaich, A., Roudies, O., Souissi, N., Salinesi, C. Selecting SPL modeling languages: A practical guide, *Third World Conference on Complex Systems (WCCS)*, Marrakech, Morocco (2015)
DOI: 10.1109/ICoCS.2015.7483312

III. Communication dans des Doctorales :

10. Achtaich, A., Salinesi, C., Mazo, R Roudies, O., Souissi, N. Multi-instance Monitoring and Configuration. Université Paris 1 Panthéon Sorbonne - CRI (2017)
11. Achtaich, A., Salinesi, C., Mazo, R Roudies, O., Souissi, N. Muti-Instantiation Of Mobile Devices Fleets. Université Paris 1 Panthéon Sorbonne - CRI (2015)
12. Achtaich, A., Roudies, O., Souissi, N. Tendances du développement mobile, *Journées Doctorales en Technologies de l'Information et de la Communication (JDTIC)*, Rabat, Morocco (2014)

Liste des abréviations

Abréviation en Français	
A/H	Automne/Hiver
CdE	Coupure d'Électricité
IDM	Ingénierie Dirigée par des Modèles
IdO	Internet des Objets
IS	Ingénierie Système
LdP	Lignes de Produits Logiciels
LdPD	Lignes de Produits Logiciels Dynamiques
MA	Modèle d'apprentissage
MC	Modèle de Configuration
MCA	Modèle de Contexte d'Application
MCD	Modèle de Contexte de Domaine
MCP	Modèle de Connaissance Publique
MP	Modèle de perception
MrC	Modèle de re-Configuration
MV	Modèle de variabilité
OQM	Objectif Questions Métriques
P/E	Printemps/Été
PdSC	Problème de Satisfaction de Contraintes
SdS_Dy	Systèmes de Systèmes Dynamiques
Abréviation en Anglais	
C	Constraint
CL	Concern Level
CP	Constraint Programming
CS	Composite State
GORE	Goal Oriented Requirepent Engineering
LGS	Landing Gear System
MAPE	Monitor Analyse Plan Execute
MC	Modèle de Configuration
REFAS	Requirement Engineering For Self-Adaptive Systems
S	State
SCT	State-Constraint Transition
SSC	Simple State-Constraint
T	Transition
V	Variabe
SPL4DSoS	Software Product Line Framework For(4) Dynamic Systems Of Systems
NoE	No Electricity
SS	Spring/Summer
AW	Autum/Winter

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1 Introduction générale

1.1. Contexte

Un système est un groupe d'éléments organisés pour accomplir un ensemble spécifique de fonctions (Sillitto et al. 2019). Au fur et à mesure que le domaine et les technologies de l'ingénierie ont évolués, la complexité des systèmes a augmenté, et leur composition et dynamique ont changé par conséquent. Ainsi, pour contrôler cette complexité, et dans l'esprit de diviser pour reigner, ces systèmes sont plutôt conçus comme des systèmes de systèmes (SdS) (Systems of Systems : SoS en anglais), constitués de sous-systèmes moins compliqués mais indépendants, répartis et hétérogènes (van Solingen et al. 2002) (Mahya and Tahayori 2016).

Par ailleurs, dans l'ère de l'Industrie 4.0, habilitée par des technologies tels que l'internet des objets (IdO) (Internet of Things, IoT en anglais), le Cloud Computing et l'intelligence artificielle, l'ingénierie des systèmes converge vers la **globalisation** et l'**autonomie** des systèmes, notamment, des dispositifs, des processus, des logiciels et des exigences. La globalisation permet d'un côté de répartir les systèmes entres des SdS multiples, et d'un autre, de les connecter afin d'assurer les services requis par le SdS, impossibles à réaliser sinon (Hein et al. 2018). Cette distribution et union permettent la mise en place de solutions des plus basiques aux plus complexes, innovantes et sophistiquées. Par exemple, une consommation intelligente peut être envisagée lorsqu'un compteur est capable d'envoyer et de recevoir de l'information sur l'usage et la consommation d'eau, d'électricité ou de gaz, et de communiquer avec d'autres dispositifs tels que le réservoir d'eau, le frigo, les ampoules ou les capteurs de présence pour atteindre un objectif commun. L'autonomie permet aux systèmes de s'adapter individuellement et collectivement (Weyns and Andersson 2013) (Saleh and Abel 2018). Cette aptitude est rendue possible par la capacité des systèmes courants de détecter, voir comprendre leurs contextes, et de le communiquer en temps réel. De la sorte, le système de gestion de consommation peut, et sans aucune intervention humaine, être reconfiguré et reprogrammé à distance.

Au bout du compte, tout comme les organes d'un corps, les systèmes de systèmes, aujourd'hui dynamiques, collaborent, échangent, communiquent et synchronisent leurs capacités, et ce dans un but précis. Des services innovants deviennent ainsi possibles. Dans le secteur de la domotique par exemple, connecter des ampoules, des interrupteurs,

des rideaux et des capteurs de mouvement et de lumière permet de prendre en charge l'aération et la luminosité d'une chambre ou d'une maison (Moreno et al., 2015; Kamienski et al. 2019; Zhang et al. 2017), selon les préférences des habitants en termes de bien-être, d'écologie et de dépenses. Dans le secteur de l'automobile, des capteurs de distance, des caméras et des GPS automatisent le stationnement d'un véhicule dans un parking (Harper 2003). Dans le secteur de l'agriculture, des capteurs d'humidité et de pluie, ensembles avec un système d'irrigation et un compteur intelligent, peuvent prendre des décisions sur la fréquence d'irrigation et quantité d'eau, pour un résultat optimal (Chinrungrueng et al. 2007), et en fonction des exigences de l'agriculteur.

Tout compte fait, la complexité des systèmes de systèmes dynamiques mariée à des exigences de plus en plus diversifiées et variables, nécessitent des mécanismes de spécification formelle et de génération de configurations qui soient valides pour des contextes divers.

1.2. Spécificités des systèmes de systèmes dynamiques

Un système de systèmes dynamiques (SdS_Dy), comme son nom l'indique, est un groupement de systèmes doté d'un ensemble de capacités qui permettent d'atteindre un objectif ultime, tout en s'adaptant aux divers bouleversements de son contexte. Ces capacités sont importantes à comprendre (Gorod et al. 2008) (Hein et al. 2018), afin de déterminer le périmètre des systèmes de systèmes dynamiques, et cerner les principaux motifs d'adaptation.

- **Multi-utilisateurs :** Les systèmes de systèmes sont généralement conçus pour des utilisateurs hétérogènes, qui ont des besoins tout aussi divergents. Un SdS_Dy conçu pour des villes intelligentes doit être correctement déployé, qu'il s'agisse d'une instanciation à Paris ou à Bangalore. Pareil pour un SdS_Dy dans le domaine domotique, qui doit répondre aux préférences des parents, des enfants et même des invités. Ainsi, est-il question de spécifier des exigences pour différents scénarios d'utilisation, avant et même après instanciation, ceci dans le but de prévoir les adaptations nécessaires à chaque groupe d'utilisateur.
- **Multi-appartenances :** Les SdS_Dy contemporains sont capables de collecter, partager et diffuser des données avec tout autre système qui soit configuré pour communiquer avec eux. Ainsi, certains systèmes peuvent appartenir simultanément

et consécutivement à plusieurs SdS_Dy, pour satisfaire un même objectif, ou même pour satisfaire des besoins différents. Un smartphone par exemple est capable d'appartenir à un SdS_Dy de parking, comme il peut appartenir à un SdS_Dy de suivi médical. Dans la première, il offre des services tels que la localisation de l'utilisateur ou la proximité à des espaces de parking. Dans la deuxième, il peut servir de capteur de signaux vitaux, ou de déclencheur d'alerte en cas de malaise.

- **Collaboration :** Grâce à leur capacité de communication, les sous-systèmes interagissent entre eux, afin de contribuer à l'aboutissement d'un objectif particulier. Il en découle qu'un changement au niveau d'un sous-système, quelle que soit sa nature (activation, désactivation, reconfiguration), affecte ceux qui l'entourent. Le comportement global du SdS_Dy est fortement lié à celui des systèmes le composant. Par conséquent, bien qu'une adaptation concerne un système en particulier, le SdS_Dy est susceptible de se reconfigurer complétement.
- **Autonomie :** Bien qu'un système (ou un sous-ensemble de systèmes) soit une partie de l'ensemble qu'est le SdS_Dy, il constitue une unité à part entière, conçue et maintenue individuellement, pouvant fonctionner indépendamment du reste des systèmes. Il est ainsi capable de fournir de la valeur ajoutée de manière isolée. Chaque système peut s'adapter individuellement, sans affecter le comportement du reste des éléments du SdS_Dy.
- **Évolution :** De nouveaux dispositifs, protocoles, standards, ou langages sont de plus en plus disponibles et ouverts, et ce, au service des concepteurs des systèmes de systèmes et des utilisateurs finaux qui sont aujourd'hui des acteurs actifs dans l'évolution de tels systèmes. Ainsi, l'évolution d'un constituants (dispositifs, élément embarqué, composant logiciel …) nécessite ou peut déclencher une reconfiguration de l'ensemble.

1.3. Problématiques de recherche

Les exigences sont au cœur de tout système informatique (Chakraborty et al. 2012), y compris les systèmes de systèmes dynamiques. Une bonne implémentation du processus d'ingénierie des exigences est fondamentale pour la réalisation de systèmes cohérents et représentatifs des besoins de ses utilisateurs. Dans le cas de SdS_Dy, après l'élicitation et l'analyse des exigences, il s'agit de spécifier les exigences propres au SdS global, ainsi que

celles de tous les sous-systèmes qui le composent, dynamiquement. Cette spécification a pour objet d'établir ce que le SdS_Dy doit faire et les conditions sous lesquelles il doit opérer. Cependant, plusieurs problématiques se posent. Elles sont relatives à la typologie d'exigences qui est très diversifiée, à la représentation d'informations hétérogènes depuis divers points de vue et à l'incapacité des mécanismes existants à configurer dynamiquement les SdS_Dy. Ces divers problèmes sont discutés plus amplement dans les sections suivantes, et les principales questions de recherches abordées dans cette thèse sont déduites en conséquence.

1.3.1. Variabilité dans la typologie des exigences

Dans l'ère où l'humanité délègue des fonctions d'analyse et de prise de décisions aux systèmes informatiques, il est important que ceux-ci estompent réellement les frontières entre machines et comportement humain, les premiers étant riches de ressources computationnelles, capables de capturer, d'analyser et de traiter des données précises en temps réel, et le deuxième, étant unique dans ses préférences, flexible, réactif et adaptable. Par conséquent, les exigences exprimées à l'égard des systèmes de systèmes dynamiques surpassent souvent les aspects fonctionnels et non fonctionnels (Dar et al. 2018). Au-delà de cela, et afin d'être à la mesure de leurs promesses, les SdS_Dy doivent généralement répondre à des exigences beaucoup plus complexes, qui prennent en compte les choix différents des utilisateurs, les réactions aux changements susceptibles de se produire, l'incertitude, etc.

De la sorte, vue la nature très interactive et évolutive des solutions et applications du domaine des SdS_Dy, il est impensable de concevoir un système de systèmes figé en termes d'exigences. Ceci est la conséquence d'une évolution dans le temps et dans l'espace comme il est détaillé ci-dessous :

- **Évolution dans l'espace:** les SdS_Dy sont généralement conçus pour des groupes d'utilisateurs, dont les besoins varient et évoluent. Un SdS_Dy dans le domaine de la domotique par exemple devrait garantir le confort de ses occupants, mais aussi de ses invités et de son personnel.
- **Évolution dans le temps :** Le principe d'un SdS_Dy est de connecter des systèmes qui, en communiquant et en collaborant, permettent la mise en place de services innovants. Ainsi, pour rester à la pointe de la technologie, et ce dans un domaine en pleine croissance, un SdS_Dy devrait être flexible et extensible, afin de bénéficier des

facultés offertes par de nouvelles générations d'outils technologiques. D'autre part, si l'on reconsidère l'exemple de la maison intelligente, les occupants sont aptes à évoluer concrètement, comme par exemple l'arrivée d'un nouveau membre de la famille ou le passage d'un enfant à l'âge adulte. Ainsi dans le premier cas, les exigences du nouveau membre doivent être prises en compte en termes de spécifications en plus de ce qui existe par défaut. Dans le deuxième cas, l'évolution morale du jeune adulte doit pouvoir se projeter dans de SdS_Dy en modifiant son profil d'exigences vers un nouveau plus adapté à ses nouveaux besoins.

Finalement, rédiger des exigences pour des systèmes de systèmes dynamiques suit une typologie particulière, représentative des besoins de plus en plus sophistiqués des utilisateurs d'un côté, et de la variabilité intrinsèque à ce type de systèmes d'un autre. La typologie d'exigences fait l'objet de la première question de recherche de cette thèse.

QR1 : Quelles sont les spécificités des systèmes de systèmes dynamiques en termes d'exigences ?

1.3.2. Représentation d'une multiplicité de points de vue

Les SdS_Dy sont puissants et innovants, mais hétérogènes et complexes (Singh et al. 2014) (Xu et al. 2014) (Union 2012). Il doivent ainsi répondre à des exigences très spécifiques. Tout d'abord, les systèmes formant un SdS_Dy comportent une disparité et hétérogénéité à plusieurs niveaux, comme c'est le cas de l'internet des objets par exemple, où des systèmes d'exploitation et des protocoles de communication hétérogènes introduisent de la variabilité dans un niveau physique, logiciel, et réseau (Baccelli et al. 2018). Ensuite, les SdS_Dy interagissent fortement avec l'environnement qui les entoure. Certains systèmes (e.g : les capteurs) sont à l'écoute des phénomènes en temps réel (température, présence, distance, nombre de pas …), d'autres (e.g : les actuateurs) sont capables de réagir à ces informations afin de planifier une tâche, corriger une anomalie ou optimiser un problème. Ainsi, le contexte et l'environnement sont des dimensions importantes à prendre en compte dans la spécification des SdS_Dy.

De plus, contrairement aux systèmes classiques, les SdS_Dy sont conçus pour satisfaire les besoins d'un ensemble d'utilisateurs qui souvent, expriment des besoins différents, des fois même divergents. Ainsi, grâce aux progrès dans le domaine de l'intelligence artificielle, il est aujourd'hui possible de capitaliser des retours des différents utilisateurs

pour détecter des schémas d'utilisation, anticiper des besoins et même proposer de nouveaux services sur mesure. Finalement, étant dans un secteur en pleine croissance, des dispositifs nouveaux, plus innovants et plus intelligents que leurs prédécesseurs sont rapidement introduits sur le marché. Ainsi, la susceptibilité et la rapidité d'évolution dans un SdS_Dy est largement plus imposante qu'il n'est le cas dans des systèmes plus classiques.

De ce fait, la conception d'un système de systèmes dynamique se base souvent et forcément sur une variété de modèles, appartenant à des paradigmes différents, traitant des points de vue distincts et impliquant des parties prenantes qui interviennent à des niveaux de maturité discordants, avant et après l'exécution. Les propriétés et concepts fondamentaux pour spécifier des SdS_Dy, ainsi que les relations qui régissent leur dynamique doivent être définis et représentés pour garantir la compréhension globale de ce domaine. Ainsi est-il primordial de répondre à la deuxième question de recherche (QR2).

QR2 : Quel cadre de travail peut servir de base conceptuelle pour la spécification des systèmes de systèmes dynamiques ?

1.3.3. Besoin en mécanismes de spécification évolutifs

Spécifier un SdS_Dy de bout en bout nécessite des mécanismes qui prennent en compte les spécificités de ce paradigme, notamment la représentation de la variabilité – caractéristique intrinsèque de ces systèmes– selon différents points de vue et le raisonnement sur les exigences dynamiques et évolutives (Hotz et al. 2019). Chacune de ces caractéristiques introduit une complexité qui n'est pas gérée par les langages de spécification actuels. La gestion de la variabilité fait l'objet d'une multitude d'approches, de manière dédiée pour certaines et de manière moins spécifique pour d'autres. Dans les deux cas, spécifier les exigences de configuration complexes telles que les exigences de composition, d'optimisation, de multi-instanciation ou de temporalité est insuffisant. Des mécanismes plus efficaces et représentatifs des besoins des utilisateurs sont dorénavant nécessaires.

De surcroît, les approches pour la gestion de la dynamique et l'évolution de tels systèmes font souvent référence aux changements de configurations qui ont lieu suite à des altérations du contexte ou de l'environnement. Cependant, les exigences elles-mêmes

sont elle aussi sujettes au changement et nécessitent tout autant des mécanismes qui représentent cette problématique est traduite par la troisième question de recherche (QR3).

> **QR3 :** Quel langage employer pour la spécification des exigences des systèmes de systèmes dynamiques ?

1.3.4. Besoin en mécanismes de configuration dynamique

La configuration d'un SdS_Dy correspond à un état qui satisfait parfaitement ou partiellement les exigences de différentes parties prenantes du système de systèmes. Lorsque les exigences sont dynamiques et progressives, les configurations le sont aussi. La configuration est une activité complexe. En effet, les systèmes composants le SdS_Dy sont multiples et fortement imbriqués dans leur structure et dans leur comportement. Ainsi, la reconfiguration d'un système peut avoir des conséquences propagées sur le reste du SdS_Dy. D'autre part, les différents modèles nécessaires pour spécifier un SdS_Dy depuis une multitude de points de vue sont corrélés. En conséquence, des décisions prises sur certains modèles doivent être considérées par les autres. Ces mécanismes doivent réduire la disparité entre des modèles hétérogènes et permettre d'effectuer des opérations d'analyse, afin de générer des configurations qui garantissent la satisfaction des exigences dynamiques.
Sur cette base, une quatrième question de recherche est formulée (QR4).

> **QR4 :** Quels mécanismes permettent de contrôler la configuration dynamique des exigences des systèmes de systèmes dynamiques ?

1.4. Hypothèses de recherche

Cette thèse s'appuie sur l'approche d'ingénierie des lignes de produits dynamiques pour spécifier des systèmes de systèmes dynamiques, ainsi que sur la transformation de modèles en problèmes de contraintes pour générer les reconfigurations dues à la dynamique dans l'espace et dans le temps. De ce fait, les hypothèses suivantes sont validées au cours des recherches réalisées :

1. La typologie d'exigences pour des systèmes de systèmes dynamiques est particulière, car elle comporte en plus des exigences pour la spécification des systèmes classiques, des exigences de variabilité de domaine, d'application et d'adaptation. Cette hypothèse correspond à la première question de recherche.
2. Le cadre de travail de l'ingénierie des lignes de produits logiciels dynamiques peut être adapté, afin de servir de base de travail pour la spécification des systèmes de systèmes dynamiques. Cette hypothèse a pour objet de répondre à la deuxième question de recherche.
3. La spécification des exigences dynamiques se ramène à la spécification d'états de configurations dans un graphe d'état-contraintes transitions (State-Constraints Transitions en anglais : SCT). Ceci correspond à la réponse à la troisième question de recherche.
4. La transformation des graphes d'états-contraintes transitions (SCT) en problème de contraintes dynamique est une approche efficace pour automatiser le processus de reconfiguration et d'analyse.
5. L'outillage du langage sur Xtext est utile pour automatiser les tâches d'analyse et de reconfiguration. Les deux dernières hypothèses correspondent à la quatrième et dernière question de recherche.

1.5. Méthodologie

Afin de répondre aux questions de recherches citées plus haut et valider les hypothèses qui en résultent, une méthodologie de recherche systématique est adoptée dans cette thèse. Il s'agit de la méthode de science de la conception (Design science) qui a pour objet de guider, valider et présenter la création d'artefacts innovants, selon un processus approprié, rigoureux et efficace, pour répondre à des problèmes organisationnels réels (Hevner et al. 2004). L'objectif commun des différents cadres méthodologiques de type « Design Science » est de décomposer le problème de recherche et conception par le biais de résolution de sous problèmes imbriqués, notamment la méthode des trois cycles (Hevner 2007), adoptée dans cette thèse, comme illustrée dans la Figure 1-1.

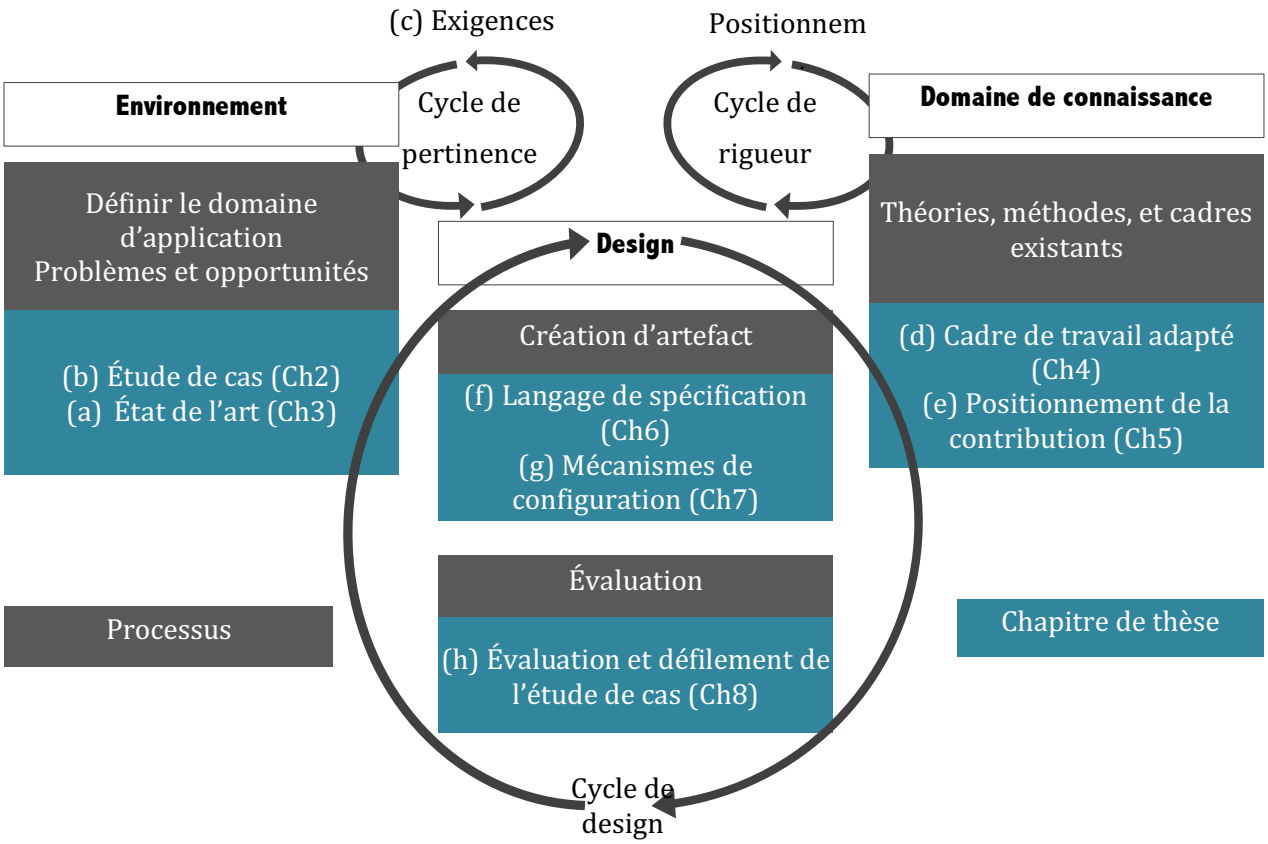

Figure 1-1 : Méthodologie de thèse selon le processus des 3 cycles du Design Science

- **Cycle de pertinence :** Comme il est illustré dans la Figure 1-1, une étude de cas détaillée est présentée. Elle énonce *le cas d'un système de systèmes dynamique dans le domaine de l'irrigation* (b) et répertorie la *collection des exigences* (c) qui en découlent pour en définir la nature et la particularité. Cette thèse expose ensuite un *état de l'art* (a) sur les approches de configuration dynamique des SdS_Dy, et ce afin de définir le périmètre de la contribution. Ceci correspond bien aux processus et aux objectifs du *cycle de pertinence*, dans lequel l'environnement concerné par le nouvel artefact est étudié, afin de définir l'impact et le périmètre de la solution désirée. Ce cycle est généralement achevé par l'exploration d'un environnement d'application réel pour capturer les exigences fondamentales de l'artefact, qui serviront aussi de base pour sa validation.
- **Cycle de rigueur :** Sur la base du cycle de pertinence, un *cadre de travail* (d) approprié est choisi et adapté. Il sert de fondements pour le positionnement de la contribution de cette thèse, et ce, par rapport aux *travaux existants* (e). Il s'agit ici du cycle de rigueur, qui consiste à positionner l'artefact dans un domaine de connaissance et à l'incorporer avec des artefacts existants.

- **Cycle de Design :** Finalement, la contribution est présentée, dans ces diverses facettes. Elle consiste d'abord en l'introduction d'un *langage pour la spécification des SdS_Dy* (f), ainsi que des *mécanismes pour automatiser les configurations dynamiques* (g). Ce dernier est ensuite *outillé* (h) pour démontrer son applicabilité et prouver sa plus-value. Ceci est le cœur du cycle de design, qui est principalement concernée par la création, l'évaluation et le raffinement des nouveaux artéfacts.

1.6. Contributions de la thèse

Dans le but d'aborder les problématiques annoncées, cette thèse présente une approche pour la spécification et la génération de configuration des SdS_Dy. Les contributions originales qui ont été réalisées sont les suivants:

Contr 1. Une typologie d'exigences spécifique aux SdS_Dy.

Contr 2. Un cadre de travail adapté à la conception des SdS_Dy.

Contr 3. Un langage pour la spécification des exigences des SdS_Dy (SCT). Le langage proposé puise ses fondements des graphes de transition d'états, augmenté de logique par contraintes, pour combiner puissance d'expression et facilité d'analyse.

Contr 4. Une approche par contraintes pour automatiser la génération de configurations dynamiques.

Contr 5. Un outillage et une démonstration du modèle sur le cas d'un système d'irrigation.

La liste des publications réalisées durant cette thèse est détaillée dans le Tableau 1-1.

1.7. Organisation du mémoire

Le Chapitre 1 décrit le contexte étudié dans le cadre de cette thèse, en l'occurrence, les systèmes de systèmes dynamiques. Les principaux problèmes rencontrés lors de la spécification de tels systèmes sont discutés et les questions de recherche, ainsi que les hypothèses qui en découlent sont introduites. L'approche design science est ensuite présentée, étant la méthodologie adoptée dans cette thèse. Les contributions sont organisées selon le Framework des trois cycles et la liste des publications est énoncée.
La suite du mémoire est organisée en trois parties qui correspondent à la méthode design science.

Référence	Article	Nom de la conférence/Journal	Date	Indexé par
Conférences				
(Salinesi et al. 2020)	Innocrowd, A Contribution to An Iot Based Engineering Product Development	COSIT'20	January'20 - Zurich	EBSCOhost
(Achtaich et al. 2019)	Evaluation of the State-Constraint Transition Modelling Language: A Goal Question Metric Approach	SPLC'19	Septembre '19 - Paris	ACM DL, Scopus
(Achtaich et al. 2017)	Designing a Framework for Smart IoT Adaptations	Africatek'17	Mars'17 - Marrakech	DBLP, Scopus, Springer Link DL
(Achtaich 2016)	Vers une plateforme de monitoring de flottes d'objets connectés	Inforsid'16	Juin'16 - Grenoble	Scopus
(Achtaich et al. 2015)	Selecting SPL modeling languages: A practical guide	WCCS'15	Novembre' 15 - Marrakech	IEEE Xplore DL, Web of Science
Revues				
(Achtaich et al. 2021)	Guidelines for the Specification of IoT Requirements: The case of smart cars.	IGI Books	January'21	Scopus, Web of Science
(Achtaich et al. 2019)	Management Capabilities for Fleets of Mobile and IoT Devices - An Evaluation Framework	IJEAT	Août'19	Scopus
(Achtaich et al. 2019)	A constraint-based approach to deal with self-adaptation: the case of smart irrigation systems	IJISA	Août'19	Clarivate, Scopus, EBSCOhost
(Achtaich et al. 2018)	A DSPL Design Framework for SASs: A Smart Building Example	EAI Endorsed Transactions	Mars'18	European Union Digital Library

Tableau 1-1 : Liste des publications réalisées dans cette thèse

La première partie est consacrée à l'étude du domaine de connaissance, et comprend les chapitres 2 et 3. La deuxième partie présente le positionnement de la contribution de cette thèse dans un cadre de travail. Elle regroupe les chapitres 4 et 5. Finalement, la troisième partie présente la contribution, et se compose des chapitres 6, 7 et 8.

Le Chapitre 2 présente le cas d'étude d'exploration ; à savoir un SdS_Dy composé d'objets connectés pour l'irrigation d'un champ d'agriculture. Un cahier des charges regroupe et classifie ensuite les exigences élucidées.

Le Chapitre 3 étudie les approches de spécification des SdS_Dy. Un état de l'art est donc dressé, présentant les principales contributions existantes, leurs limites et perspectives d'extension.

Le Chapitre 4 présente le cadre de travail. Dans ce chapitre sont définis les concepts clés du domaine d'étude, ainsi que les caractéristiques, les parties prenantes, les modèles, les flux d'information et les différents processus de configuration.

Le Chapitre 5 introduit les notions fondamentales à l'élaboration de la contribution. Il positionne les différents aspects liés à la spécification selon le cadre proposé, et introduit les mécanismes pour automatiser la génération des configurations valides.

Le Chapitre 6 est au cœur de la contribution de cette thèse. Il introduit le langage State Constraint Transition (SCT) pour la spécification formelle de systèmes de systèmes dynamiques en graphes de contraintes, notamment en présentant sa syntaxe et sa sémantique. Le chapitre illustre le langage à travers une application sur le cas du système d'irrigation

Le Chapitre 7 présente les règles de transformation de graphe SCT, en programmes de contraintes. Différentes techniques de ce paradigme sont introduites pour répondre à divers besoins de spécification. L'outillage du langage SCT sur Xtext est ensuite présenté, et les configurations générées sont analysées.

Le Chapitre 8 évalue le pouvoir d'expression, le passage à l'échelle et l'indépendance de domaine du langage proposé, suivant une approche Objectif-Questions-Métriques, ainsi que l'efficacité perçue des graphes, évalués selon les principes de la théorie perceptive.

Le Chapitre 9 récapitule les principaux résultats obtenus dans le cadre de cette thèse et propose les pistes d'amélioration éventuelles comme perspectives.

Partie I

Modélisation des systèmes de systèmes dynamiques : Vue d'ensemble

Chapitre 2 Étude des exigences : Le cas GreenLife

2.1. Introduction

Le cas de la solution GreenLife, un SdS_Dy d'irrigation a été retenu afin d'illustrer le concept de systèmes de systèmes dynamiques, c'est-à-dire d'une composition de systèmes qui communiquent, s'activent ou se désactivent à la volée, individuellement ou en groupe, en fonction du contexte. Les cas d'utilisations et les différents scénarios d'usage sont inspirés d'une étude provenant du domaine de l'agriculture (V. Elsner et al. 2000; Waaijenberg 2006; Rajalakshmi and Devi Mahalakshmi 2016; Kamienski et al. 2019).

Contrairement aux études de cas de validation, qui se préoccupent de vérifier une théorie, les études de cas d'exploration contribuent à définir des théories, dans leur contexte humain et social (Klein and Myers 1999). Cette pratique, fréquemment utilisée dans la recherche (Anda et al. 2006)(Sewell et al. 2018), est exploitée dans ce chapitre pour identifier les besoins des utilisateurs du SdS_Dy GreenLife, les généraliser, et en dégager une typologie qui puisse être employée pour tout type de SdS_Dy.

Le cas décrit en premier lieu une solution « générique », c'est-à-dire qui s'adresse à différents usages avec des utilisateurs exprimant des besoins plus ou moins spécifiques. Il introduit par la suite le cas de Maria, une utilisatrice qui souhaite concevoir et installer un SdS_Dy d'irrigation auto-adaptable, afin de maintenir une performance satisfaisante dans un environnement particulier décrit dans le chapitre. Ce cas illustre la nature dynamique des exigences d'un même utilisateur, amenant à concevoir un système de systèmes qui doit s'adapter dynamiquement au contexte, et motive la nécessité d'une démarche de spécification bien particulière pour ce type de système. Le chapitre est conclu en déduisant une typologie d'exigences, qui démontre qu'en plus de la complexité liée aux problématiques d'adaptation, les exigences des SdS_Dy, sont-elles mêmes diverses, complexes et dynamiques.

2.2. La solution GreenLife

Se situant dans un pays à fort potentiel agricole, GreenLife Solutions est une société qui a pour objectif de créer un produit innovant au service du marché agricole. Les fondateurs souhaitent mettre en place un « système d'irrigation intelligent » sous forme d'une solution du domaine de l'IdO qui surveille les changements environnementaux et

maintient les niveaux requis d'humidité, de température et de luminosité, de manière à automatiser complétement le processus d'irrigation. Suite à une étude de marché, GreenLife constate que les besoins des agriculteurs varient pour de nombreuses raisons au rang desquelles l'efficacité, les coûts, et les préférences personnelles. Dès lors, la société anticipe la diversité des situations d'usage, et décide de proposer une solution paramétrable qui permette d'offrir une diversité de configurations tout en garantissant un certain niveau de qualité de service. Leur portfolio contient des constituants saillants parmi lesquels on distingue : des irrigateurs, une toiture automatique, des actuateurs, des sources d'eau et des capteurs, comme l'illustre la Figure 2-1.

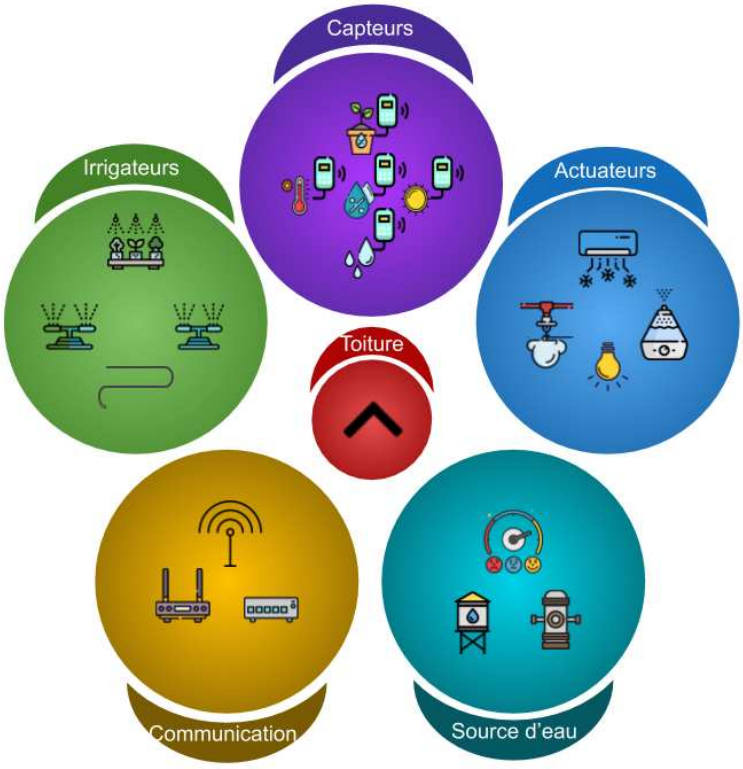

Figure 2-1 : Les constituants du système d'irrigation GreenLife

2.2.1.Irrigateurs

La solution GreenLife permet d'assurer l'irrigation par le biais d'arroseurs (Sprinkler), d'une conduite compte-gouttes (Dripline), ou à l'aide d'un centre Pivot (CenterPivot). Chacun de ces dispositifs peut être installé en plusieurs instances (pour le centre Pivot, c'est le nombre de diffuseur qui varie), selon la taille et la topologie du champ à irriguer. Bien évidemment, le SdS_Dy d'irrigation doit comporter au moins un de ces dispositifs afin de assurer la tâche d'irrigation qui est essentielle. Cependant, il est aussi possible

d'installer simultanément plusieurs dispositifs, notamment pour assurer d'autres objectifs, comme l'efficacité énergétique, une utilisation parcimonieuse de l'eau, un arrosage adapté aux besoin des plantes cultivées, ou la rapidité de l'irrigation. Chacun de ces dispositifs d'irrigation possède des caractéristiques qui lui sont propres :

Les Sprinkler peuvent pivoter de 360°, 240°, 180° ou 90°, et l'eau peut s'échapper avec une pression entre 20 et 60 psi (Pressure per square inch).

Les Dripline sont disponible en différentes longueurs. La pression d'eau varie entre 15,30 et 40 psi pour un nombre d'émetteurs égal à 114, 200 ou 232.

Un CenterPivot est composé d'une suite de Pivot_Sprinkler, dont le nombre est choisi en fonction du rayon du champs concerné. Il réalise des rotations variant de 90° et 360°, avec une pression respective de 30 à 80 psi. Le Figure 2-2 résume la composition du dispositif d'irrigation. Les éléments dont la valeur est paramétrable sont annotés avec une étoile (*) et les éléments multi-instanciés sont annotés avec un dièse (#).

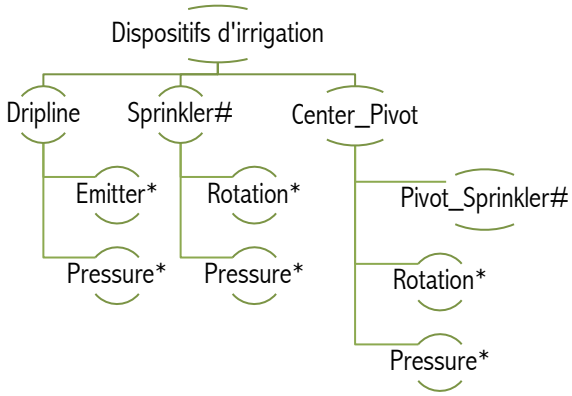

Figure 2-2 : Dispositif d'irrigation du système GreenLife

2.2.2.Toiture

Outre le dispositif d'irrigation, une toiture (Rooftop) peut également être ajouté au SdS_Dy d'irrigation, principalement pour protéger le champ des intempéries et y conserver la chaleur nécessaire à la protection des plantes cultivées, tout en contribuant à l'irrigation lorsqu'il pleut (grâce à un système d'ouverture), et en contrôlant la luminosité du champ. Pour contrôler la luminosité du champ, et ainsi assurer que les besoins des plantes sont satisfaits, la toiture employée peut être transparente (Glass) ou opaque (Shader), étant donné que certaines plantes s'épanouissent lorsqu'elles sont exposées à une source de lumière (Aloe Vera, tomates, etc), quand d'autres ont plutôt besoin d'ombre (champignons, endives, menthe, etc). Si l'on choisit un Rooftop mobile, il

est alors possible de contrôler son degré d'ouverture et de fermeture (Close, Open, Partial) Le Figure 2-3 présente la composition du Rooftop.

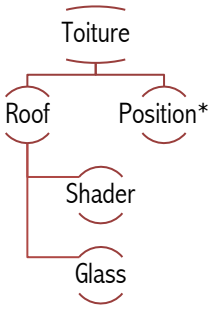

Figure 2-3 : Éléments de toiture du système GreenLife

2.2.3.Actuateurs

Un ensemble d'actuateurs peut être installé pour contrôler les autres paramètres influents, notamment la température, la luminosité et l'humidité.

Ainsi, des régulateurs de température (Air_Conditioner) peuvent être installés en plusieurs instances, pour maintenir la température souhaitée. Ils opèrent en plusieurs modes (Fan, Cooling, Heating) et peuvent être configurés pour garantir des températures variant de 5° à 40°, avec des vitesses différentes (High, Low, Auto). Des ampoules intelligentes (Bulb) peuvent jouer double rôle : elles peuvent assurer le chauffage du champ, comme elles peuvent accroitre la luminosité. De plus, un système de pulvérisation d'eau (Mist_Dispenser) est peut aussi être employé pour abaisser les températures, ou pour augmenter l'humidité de l'air, à l'aide de brouillard d'eau. Finalement, des humidificateurs (Air_Humidifier) peuvent être installés pour contrôler l'hygrométrie du champs. Le Figure 2-4 présente la composition des dispositifs d'actuation.

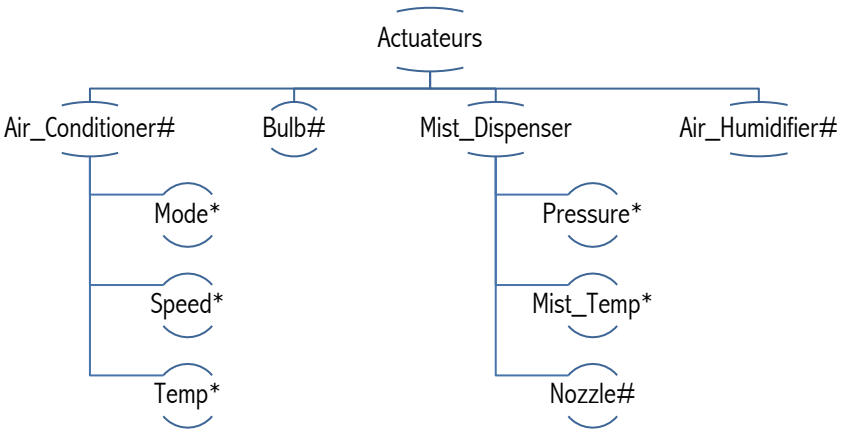

Figure 2-4 : Actuateurs du système GreenLife

2.2.4.Source d'eau

Les dispositifs qui assurent l'arrosage et la régularisation de température, respectivement, les Sprinkler, le Dripline, le Center_Pivot d'un côté, et le Mist_Dispenser et l'Air-Humidifier d'un autre, peuvent être alimentés par deux sources d'eau: l'approvisionnement via un distributeur régional (Mainswater), ou bien l'eau de pluie collectée et conservée dans un réservoir personnel local (Rainwater). Un dispositif d'adjonction de produits fertilisants (Fertilizer_Unit) peuvent installés au niveau des distributeurs d'eau afin distribuer le fertiliseur lors de l'arrosage. Un compteur intelligent peut être connecté aux deux sources, afin de mesurer le niveau d'eau consommé ainsi que le niveau d'eau disponible en réservoir (dans le cas où l'eau de pluie est utilisée). En outre, une pompe est nécessaire pour extraire l'eau du réservoir lorsque celui-ci est utilisée. Enfin, un système d'alarme (Threshold_Alarm) peut éventuellement être installé pour signaler les consommations de pointe. Le Figure 2-5 présente la composition du système GreenLife concernant la gestion des sources d'eau.

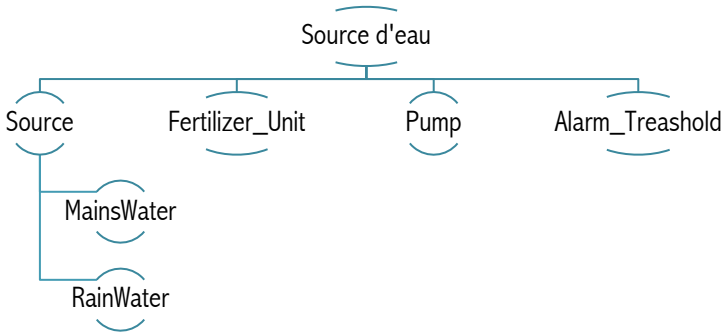

Figure 2-5 : Sources d'eau du système GreenLife

2.2.5.Capteurs

Le système GreenLife offre un dispositif de collecte d'informations sur l'état général du terrain. Plusieurs capteurs peuvent être installés sur site offrant ainsi une variété de mesures : les capteurs d'humidité, les capteurs de pluie, les capteurs de lumière, et les capteurs de température. Les capteurs d'humidité (Humidity_Sensor) permettent de collecter des informations sur l'humidité du sol (Sensor – Soil), et de l'air (Sensor – Air). Un Humidity_Sensor peut être en état d'hibernation jusqu'au besoin (Slave) ou actif (Master). De plus, au moins un capteur d'humidité en mode Master doit être actif dans le SdS_Dy d'irrigation. Certains capteurs d'humidité comportent des alarmes ou des

ampoules intégrées pouvant sonner ou clignoter. Ces derniers ne peuvent être actifs que lorsque le capteur est en mode `Master`. Un capteur de pluie (`Rain_Sensor`) peut être installé à l'extérieur pour signaler la tombée de pluie, soit de manière instantanée, soit sur une durée déterminée (`TimeSpan`). Enfin, les capteurs de lumière (`Light_Sensor`) et de température (`Temperature_Sensor`) mesurent la luminosité et la température à l'intérieur de la serre. et éventuellement, à extérieur. Le Figure 2-6 présente la composition des capteurs.

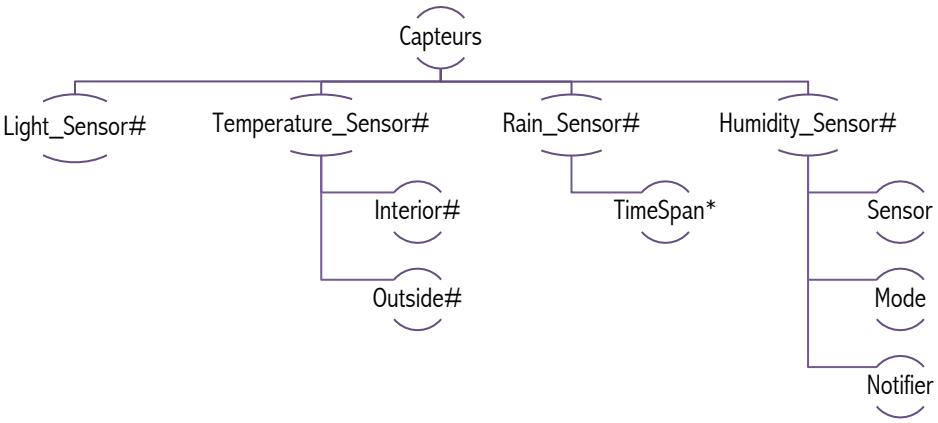

Figure 2-6 : Capteurs du système GreenLife

2.2.6.Communication

Les différents dispositifs présentés ci-dessus sont connectés et communiquent via le protocole Zigbee [1] qui supporte différentes topologies des systèmes communicants. Deux topologies spécifiques sont autorisées dans la solution GreenLife : en étoile et distribuée. La topologie en étoile repose sur un unique coordinateur Zigbee, qui récupère tous les flux d'information de manière centralisée, et est chargé de les dispatcher. La topologie distribuée requiert au moins un routeur qui récupère et réoriente les flux de données de manière plus complexe. Cette topologie plus coûteuse présente en contrepartie le double avantage d'être plus résiliente et adaptable en cas de re-disposition des terrains. Le Figure 2-7 présente la composition des dispositifs de communication.
Certains dispositifs sont également capable de communiquer via le protocole Wi-Fi, et ce, principalement pour mettre à jour les bases de données du Cloud.

[1] https://zigbeealliance.org/

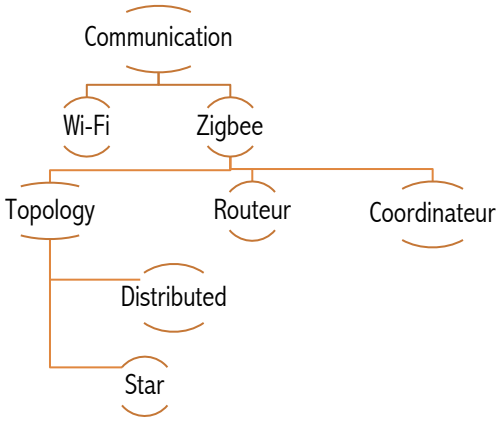

Figure 2-7 : Éléments de communication

Le Tableau 2-1 récapitule l'ensemble des dispositifs impliqués dans une solution GreenLife et que la société présente dans son « portfolio » commercial. Une multitude de SdS_Dy, composés d'une variété de dispositifs pouvant être configurés de différentes manières, peuvent être dérivés à partir de ce portfolio.

Le tableau présente pour chaque dispositif constitutif de la solution GreenLife les quantités minimales et maximales ou fixes requises. Le tableau présente aussi deux catégories d'offres commerciales. La première, intitulée « basique » correspond à la configuration que GreenLife considère comme standard et propose par défaut. La deuxième, intitulée « Mises à niveau » correspond à de nouveaux dispositifs (✓) qui complètent l'offre basique, et qui offrent des fonctionnalités plus sophistiquées, ou bien à la duplication de certains dispositifs (>1), pour assurer plus de précision et plus de flexibilité en termes de configurations. Cette mise à niveau, considérée comme plus sophistiquée, nécessite une configuration rigoureuse. Afin de réaliser cette configuration, il convient de considérer les éventuelles contraintes d'utilisation, les dépendances à entre constituants, et les adaptations requises dans certains cas d'utilisation. Ces informations sont fournies sans distinction dans la colonne « commentaires » à droite du tableau.

Portfolio GreenLife©			**Offres**		**Commentaires**
Dispositif	**Identifiant**	**Quantité**	**Basic**	**Mises à niveau**	
	Spray Sprinkler	[0..*]	1	>1	Possibilité d'adapter la pression d'eau et la rotation. Une mémoire doit être incluse pour planifier les cycles d'irrigation
	Rotor sprinkler	[0..*]			

	Dripline	[0..*]			Pour le nombre d'émetteurs 114,200, 232, les pressions respectives sont 15,30,40
	Center_Pivot	1			Possibilité d'adapter le nombre d'ajutages, la pression d'eau et la rotation
	Rooftop	1		✓	Peut être transparent, opaque, ou les deux en même temps
	Bulb	[0..*]	1	>1	Doivent être désactivés lorsque le plafond est utilisé en mode ouvert
	AC	[0..*]			Possibilité de configurer le mode, la vitesse, et la température
	Mist_Dispenser	1		✓	
	Humidifier	[0..*]		✓	Doit être accompagné d'un capteur d'humidité d'air
	MainsWater	1	1	>1	
	TankWater	1			Une pompe est nécessaire pour cette source d'eau
	SmartMeter	1		✓	
	LightSensor	[0..*]		✓	
	HumiditySensor	[0..*]	✓		Peuvent être configurés en Master ou en slave. Et peuvent contenir des systèmes de notification. Peuvent en outre mesurer l'humidité d'air
	TemperatureSensor	[0..*]		✓	
	RainSensor	[0..*]		✓	Le temps de notification peut être configuré
	AirHumiditySensor	[0..*]		✓	
	Wifi	1	✓		
	ZigbeeCoordinator	1	✓		
	ZigbeeRouter	[0..*]		✓	Sont exigées dans le cas d'une typologie distribuée

Tableau 2-1 : Portfolio GreenLife

2.3. Configuration du système GreenLife pour la cliente Maria

Maria possède une petite ferme dans la région de Doukkala, au Maroc. Cela fait quelques mois qu'elle gère sa propre Pizzeria, et souhaite planter ses tomates afin d'offrir un produit « fait maison » ce qui lui permettrait de se démarquer stratégiquement des concurrents qui utilisent des sauces ou des concentrés achetés. Cependant, les quantités nécessaires sont très importantes, et son rythme de travail à la pizzeria ne lui permettent pas d'envisager de dégager le temps nécessaire pour faire tous les déplacements nécessaires au maintien de la plantation.

La tomate est une plante qui exige 10 à 12h d'exposition à la lumière, qui s'épanouit à une température de l'air entre 20° et 30° le jour, et 15° et 20° la nuit, et lorsque l'irrigation est constante et stable. La région de Doukkala est idéale pour ce type de plantation. Au printemps et en été, les températures y varient entre 15 et 30°, et la luminosité y est idéale. Les précipitations toutefois ne sont pas régulières. De plus, entre Octobre et Mars, les températures baissent jusqu'à 5°, et en raison de pluies et de brouillards récurrents, la luminosité est faible, même pendant la journée (Sewell et al. 2018).

La solution GreenLife a été recommandée à Maria pour l'aider à superviser à distance le champs de tomates, tout en garantissant une production continue sur toute l'année, et ce malgré les conditions inconstantes du lieu de production.

Parmi les choix proposés par GreenLife, résumés en Figure 2-1, Maria opte pour une double irrigation : par Sprinkler et par Dripline. Elle installe ainsi 4 Sprinklers de type Spray, et 6 de type Rotor. Elle choisit aussi une ligne d'irrigation Dripline avec plusieurs émetteurs ajustables. Comme les tomates ne sont pas des plantes qui tirent parti de l'ombre, Maria ne sélectionne que le Rooftop de toiture transparent, avec la possibilité d'ouverture et de fermeture complète et partielle. Elle choisit également 5 ampoules (Bulbs) afin d'améliorer la température et la luminosité, selon les exigences des tomates, et 4 Air_conditioner afin de contrôler la température, notamment lorsqu'il fait trop chaud. Pour des raisons d'efficacité de la consommation d'eau, Maria choisi de combiner les deux options d'alimentation Mainswater et TankWater, ce qui permet de tirer parti des deux sources d'eau alternatives, et de prendre un compteur (Smart_Meter) pour inspecter la consommation d'eau notamment dans la perspective d'une maîtrise des coûts. Le dispositif est complété d'une pompe (Pump) pour récupérer l'eau du réservoir, et d'une alarme (Alarm_Treshold) pour notifier des niveaux d'eau limites. Afin de

mesurer les différents paramètres environnementaux, Maria choisit une variété de capteurs (Humidity_Sensor, Temperature_Sensor, Light_sensor, et Rain_Sensor). Les capteurs d'humidité ne seront utilisés que pour capturer l'humidité du sol, l'humidité de l'air étant généralement dans les normes dans la région de Doukkala.

Finalement, pour ce qui est de la communication entre les divers dispositifs, Maria -peu convaincue par l'offre plus sophistiquée, opte pour une topologie en étoile, impliquant un coordinateur Zigbee. Certains dispositifs peuvent également communiquer en Wi-Fi. La Figure 2-8 illustre l'ensemble des dispositifs sélectionnés par Maria.

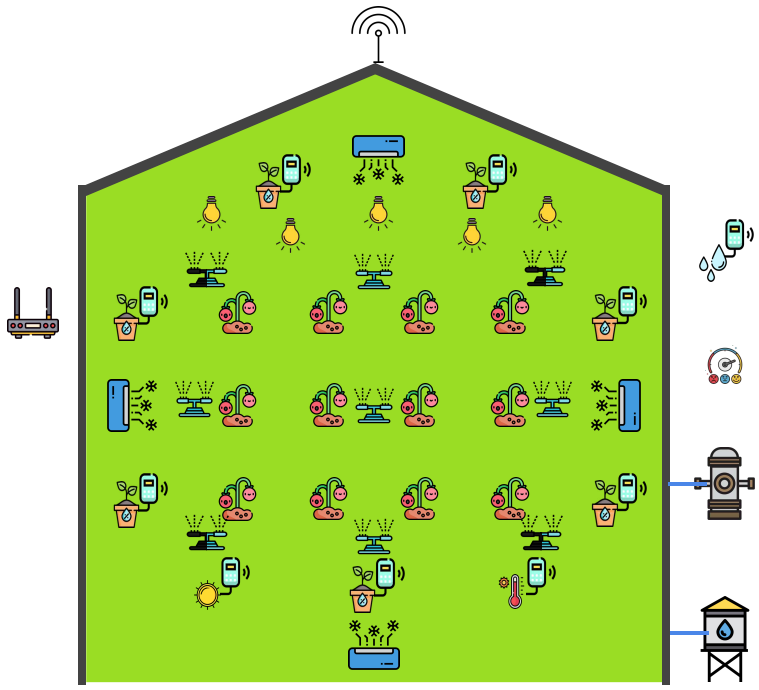

Figure 2-8: Système d'irrigation dérivé pour Maria

Bien que Maria ait choisi une composition stable de son SdS_Dy, ses attentes par rapport au fonctionnement de celui-ci, sont variables en fonction des circonstances. En effet, Maria d'un certain nombre de constats :

1. Au printemps et en été, la région de Doukkala est bien ensoleillée, et la température est exactement celle dont les tomates ont besoin pour s'épanouir. Les dispositifs du SdS_Dy d'irrigation servant à gérer la température et la luminosité jouent alors un rôle secondaire. Cependant, en vue des sécheresses très probables pendant ces deux saisons, l'irrigation doit être automatisée. La consommation d'eau est par conséquent une préoccupation très importante, et Maria exige dans ce sens un monitoring rigoureux des niveaux d'eau dans le réservoir, et une maîtrise de la consommation de l'eau provenant de la centrale, afin de gérer les coûts. Finalement, comme tous les

capteurs comportent une plaque solaire embarquée pour recharger les batteries, et qu'on peut s'attendre à ce qu'il y ait suffisamment de soleil à ces saisons, Maria espère qu'elles se rechargeront suffisamment. Interrogée à ce sujet, elle estime très faible et sans conséquence importante et durable le risque d'épuisement total des batteries.

2. En automne et en hiver, les températures peuvent descendre jusqu'à 5°, et la luminosité, même pendant la journée, n'est pas suffisante. Le SdS_Dy favorise ainsi une gestion fermée de la plantation. Le toit ne s'ouvre que lorsqu'il pleut, et que l'humidité du sol est en dessous de la normale. Ce choix de fermeture a des implications sur les niveaux de batterie des capteurs, qui se chargent lentement au moyen des lampes, et des rayons de soleil qui se font rares pendant ces saisons. L'irrigation naturelle est fréquente grâce aux pluies récurrentes, dès lors, la question de la consommation d'eau devient négligeable.
3. La région de Doukkala est une région rurale, dont les infrastructures peu fiables et vulnérables causent une panoplie de difficultés prévisibles, notamment, en termes d'infrastructure électrique. Des coupures d'électricité soudaines se produisent souvent dans la région. Maria décide ainsi de mettre en place un générateur, qui prendra la relève en cas de panne. Cependant, lorsque ceci se produit, le fonctionnement du SdS_Dy devrait se concentrer uniquement et exclusivement sur l'irrigation.

Maria définit trois modes de fonctionnement différents de son dispositif : un mode Automne/Hiver (A/H), un mode Printemps/Été (P/É), et finalement, un mode coupure d'électricité (CdE). Il est à noter que ces trois modes sont mutuellement exclusifs, mais ne suivent pas une séquence stricte, puisque la coupure d'électricité peut intervenir à tout moment, été comme hiver.

2.3.1. Mode Automne/Hiver (A/H)

Ce mode est activé entre le mois d'Octobre et le mois de Mars (A/H). Ce mode favorise une gestion fermée de la plantation, et optimise l'usage des capteurs pour éviter la déplétion des batteries. Les 15 exigences de Maria pour ce mode sont décrites dans le Tableau 2-2 selon le format standard de spécification des exigences en langage naturel (Mavin et al. 2009) :

#Exigence : [<Événement>][<Condition>]<le système> <doit/devrait/peut> <propriété>+.

ID	Exigences
E_A/H_1	Quand le niveau du réservoir est supérieur à la mesure optimale (tankLevel > OpTankWater), le SdS_Dy doit activer les Sprinkler 1, 2, 3 et 4 en mode Rotor, avec une pression 30, et une rotation 240°
E_A/H_2	Quand le niveau du réservoir est inférieur à la mesure optimale (tankLevel < OpTankWater), le SdS_Dy doit activer le compte-goutte, avec 114 émetteurs
E_A/H_3	Quand il pleut (Rain = true & HL < Normal), le SdS_Dy doit utiliser Rooftop en mode open, et doit désactiver les Sprinkler et compte-goutte
E_A/H_4	Lorsque la température est supérieure à 10° (Température >= 10°), le SdS_Dy doit activer 3 ampoules
E_A/H_5	Lorsque la température est supérieure à 5° (Température >= 5°), le SdS_Dy doit activer 5 ampoules
E_A/H_6	Lorsque la température est inférieure à 5° (Température < 5), le SdS_Dy doit activer 2 AC en mode Heat, à éjection moyenne
E_A/H_7	Quand toit est en position ouverte et la température < 20° (open=True /\ Température<20°), le SdS_Dy doit activer 4 AC, en mode Heat, et en Speed High
E_A/H_8	Quand les niveaux de batterie de tous les capteurs d'humidité sont supérieurs à 50% (BL[1]>50 /\ … BL[7]>50), le SdS_Dy doit activer 3 capteurs d'humidité en mode Master
E_A/H_9	Quand les niveaux de batterie d'au moins un capteur d'humidité sont inférieurs à 50% (BL[1]<=50 \/ … BL[7]<=50), le SdS_Dy doit activer le capteur d'humidité dont le niveau de batterie est supérieur à 20% en mode Master, le reste en slave
E_A/H_10	Quand la consommation d'électricité est normale que la luminosité est haute, ou que la consommation est haute et la luminosité est normale, à condition qu'il ne pleuve pas ou que le niveau d'humidité soit en dessous du normal, ((ConsumptionE>optimale/\Brightness>=High) \/ (ConsumptionE < optimale /\ Brightness >= Low)) /\ (rain=false\/HL<Normal), le SdS_Dy doit activer le Rooftop en position ouverte
E_A/H_11	Quand la consommation d'électricité est normale, et que la luminosité n'est pas haute, ou qu'il pleut et le niveau d'humidité du sol est en dessous de la normal (ElectricConsumption > optimale & Brightness < High & Rain=False) OR (Rain=True /\ HL>Normal) le SdS_Dy doit activer le Rooftop en position fermée
E_A/H_12	Quand le toit est en position fermée pendant une durée > 10h, le SdS_Dy doit activer les ampoules
E_A/H_13	Quand le toit est en position fermée pendant plus de 2 jours, le SdS_Dy doit activer le Rooftop en position Partielle (Partial)
E_A/H_14	Quand le niveau du réservoir est inférieur à la valeur optimale (tankLevel <= MinTankWater), le SdS_Dy doit utiliser le Mainswater comme source d'eau
E_A/H_15	Quand le niveau du réservoir est supérieur à la valeur optimale (tankLevel > MinTankWater), le SdS_Dy doit utiliser le Rainwater comme source d'eau

Tableau 2-2 : Exigences du mode Automne/Hiver (A/H)

2.3.2.Mode Printemps/Été (P/É)

Ce deuxième mode se déclenche entre le mois d'Avril et Septembre. Il favorise une plantation ouverte, et vise à optimiser l'utilisation des ressources d'eau disponibles. Les 15 exigences recensées pour le mode Printemps/Été sont décrites dans le Tableau 2-3.

ID	Exigence
E_P/É_1	Quand le niveau du réservoir est supérieur à la valeur optimale (tankLevel > opTankWater), le SdS_Dy doit activer les Sprinkler en mode Spray 1 à 9 avec une rotation 45° et une pression 60psi
E_P/É_2	Quand le niveau du réservoir est entre la valeur optimale et minimale (MinTankWater < tankLevel < opTankWater), le SdS_Dy doit activer les sprinkler 1 à 4 en mode Rotor, avec une pression 40psi, et une rotation de 240°
E_P/É_3	Quand le niveau du réservoir est inférieur à la valeur minimale (tankLevel < MinTankWater), le SdS_Dy doit activer tous les émetteurs du compte-goutte avec une pression 30psi
E_P/É_4	Quand il pleut et que le toit est ouvert (Rain=True&Open=True), le SdS_Dy doit utiliser le Rooftop pour l'irrigation
E_P/É_5	Lorsque la température est supérieure à la température maximale (Température >= 30°), le SdS_Dy doit activer le AC
E_P/É_6	Si le niveau de batterie de tous les capteurs d'humidité est supérieur à 60% (BL[1]>60 /\… BL[7]>60), ils sont tous en mode Master
E_P/É_7	Si le niveau de batterie d'au moins 1 capteur d'humidité est inférieur à 60% (BL[1<60 \/… BL[7] <60) les 3 capteurs d'humidité dont les niveaux de batteries sont les plus élevées sont en mode Master
E_P/É_8	Quand il ne pleut pas ou que le niveau d'humidité du sol est en dessous de la normale (Rain=false \/ HL< Normal), le SdS_Dy doit activer le toit en mode ouvert
E_P/É_9	Quand il pleut, et que le niveau d'humidité du sol est en dessus de la normale (Rain=True & HL>Normal), le SdS_Dy doit activer le toit en position fermée, et doit activer le light Bulb pendant 10h chaque 24h
E_P/É_10	Quand toit est activé, et la température est inférieure à 20° (open=True /\ Température <20°), le SdS_Dy doit activer le AC, en mode Heat, et Speed High
E_P/É_11	Quand le toit est désactivé pendant plus de 2 jours, le SdS_Dy doit activer le Rooftop en Opening Partial
E_P/É_12	Quand le niveau du réservoir diminue (tankLevel <= MinTankWater), le SdS_Dy doit utiliser le Mainswater comme source d'eau
E_P/É_13	Quand le niveau du réservoir s'élève (tankLevel > MinTankWater), le SdS_Dy doit utiliser le Rainwater comme source d'eau
E_P/É_14	Le SdS_Dy peut utiliser le protocole WI-FI pour connecter l'ensemble des dispositifs

Tableau 2-3 : Exigences du mode Printemps/Été (P/É)

2.3.3. Mode coupure d'électricité (CdE)

Le mode CdE est indispensable pour gérer le système de manière adaptée en cas de coupure subite d'électricité afin d'éviter les conséquences importantes qui peuvent en découler. Lorsque le système est en mode CdE, il met en veille tous les dispositifs, sauf la ligne compte-goutte, un climatiseur, et une sélection de capteurs d'humidité. Quatre exigences ont été recensées pour spécifier les adaptations à mettre en œuvre dans le mode coupure d'électricité, comme le montre le Tableau 2-4.

Les actions déclenchées au cours de ce mode ; telles que la fermeture du toit ou bien la désactivation de certains dispositifs sont maintenues tant que la coupure d'électricité persiste. Le cas échéant, et dépendamment du temps qu'il fait, ces actions sont interrompues de manière à ce que le fonctionnement du système d'irrigation redevienne optimal au contexte courant. Ceci est géré par les exigences décrites pour les deux autres modes, citées plus haut.

ID	Exigence
E_CdE_1	Lors d'une coupure d'électricité se produit (`PowerFailure = True`), le SdS_Dy doit utiliser le Dripline, avec un nombre d'émetteur égale à 114
E_CdE_2	Lors d'une coupure d'électricité se produit (`PowerFailure = True`), le SdS_Dy doit utiliser un seul capteur d'humidité, tant que son niveau de batterie est supérieur à 20%
E_CdE_3	Si le mode Automne/Hiver est actif, quand une coupure d'électricité se produit (`Month >= 4 /\ Month <10 /\ PowerFailure = True`), le SdS_Dy doit fermer le toit, et activer le AC pour régulariser la température
E_CdE_4	Si le mode Printemps/Été est actif, quand une coupure d'électricité se produit (`Month <4 Month \/ Month >= 10 /\ PowerFailure = True`), le SdS_Dy doit ouvrir le toit, et désactiver le AC pour régulariser la température

Tableau 2-4 : Exigences du mode coupure d'électricité (CdE)

De ce fait, dépendamment du contexte dans lequel se trouve le SdS_Dy de Maria, une configuration est générée. Celle-ci répond aux exigences de chaque mode, et correspond à un fonctionnement optimal du SdS_Dy. Le Figure 2-9 présente les adaptations requises pour chaque mode.

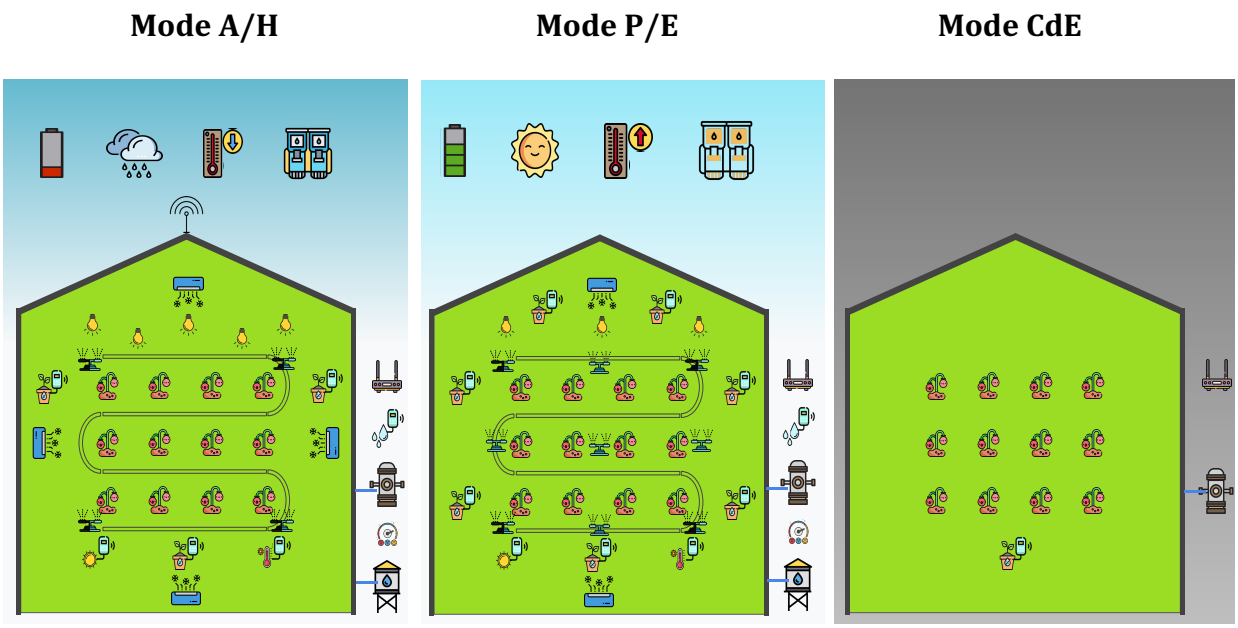

Figure 2-9 : Adaptations du SdS_Dy de la cliente Maria

Au bout du compte, une spécification efficace de la solution GreenLife doit permettre aux utilisateurs de choisir parmi une sélection de dispositifs et d'éléments qui les constituent en fonction de leurs attentes. Ce choix doit par ailleurs respecter les dépendances et les contraintes de domaine qui existent entre ces différents constituants, qu'ils soient logiciels ou matériels. En outre, afin de produire des systèmes de systèmes sur mesure et capables de fonctionner dans des contextes variables dans le temps, la spécification doit traduire des exigences variables dans l'espace et dans le temps, c'est à dire exprimant non seulement la diversité des attentes d'un client à l'autre, mais aussi le fait que pour un même client, les attentes diffèrent dans le temps, au fur et à mesure de l'évolution du contextes d'utilisation.

2.4. Une typologie d'exigences pour les SdS dynamiques

Dans le but d'élaborer une typologie représentative des exigences pour la spécification de systèmes de systèmes dynamiques, une revue de la littérature a été effectuée, en commençant à partir de la liste des exigences exprimées pour le cas du système d'irrigation GreenLife. Les premiers résultats montrent que ces exigences peuvent être classifiées selon le processus d'ingénierie qu'elles représentent. Ainsi, on retrouve des exigences relevant du domaine, de l'application ou de l'adaptation.

2.4.1. Protocole de la recherche bibliographique

La liste initiale d'exigences déduites du cas de la solution GreenLife permet de détecter quelques types d'exigences, lesquelles représentent la base de la typologie proposée. Dans un second temps, une recherche plus exhaustive est effectuée en s'appuyant sur un état de l'art. La revue de la littérature est menée en s'appuyant sur la chaine de recherche « ("Self adaptive system" OR "IoT" OR "Complex systems") AND ("requirement specification" OR "requirement elicitation" OR "requirement gathering" OR "requirement analysis") AND (Classification OR framework OR Typology OR Taxonomy) ». Seuls les articles publiés dans des journaux ou des conférences, rédigés dans les langues anglaise ou française, et qui ont été publiés depuis l'année 2010 sont alors retenus (critères d'inclusion). En plus de ces critères d'inclusion, un filtre est effectué sur la base du titre et du résumé des articles afin d'écarter les articles qui ne sont pas susceptibles de présenter des exigences de SdS dynamiques (critères d'exclusion). Les résultats de cette recherche sont présentés dans le Tableau 2-5.

Résultat initial	Filtre 1 (Titre)	Filtre 2 (EA1)	Filtre 2 (EA2)
468	111	30	20

Tableau 2-5 : Résultats et filtres de la recherche

L'objectif de l'étude étant de confirmer/compléter une liste d'exigences d'ores et déjà établie, il a été décidé que l'analyse et la synthèse des données se fasse sur un sous-ensemble des publications retenues (Turner et al. 2008). Cependant, afin de minimiser le risque de préjugé, une sélection « aléatoire » de 30 articles est faite. Cet échantillon aléatoire (EA1), disponible en Annexe 2 , et considéré représentatif, est méticuleusement étudié, pour compléter la liste des exigences. Au final, une deuxième sélection aléatoire de 20 articles (EA2) est faite depuis le lot inexploité du résultat initial de la recherche, afin de valider la typologie finale des exigences. Ce processus est illustré dans la Figure 2-10.

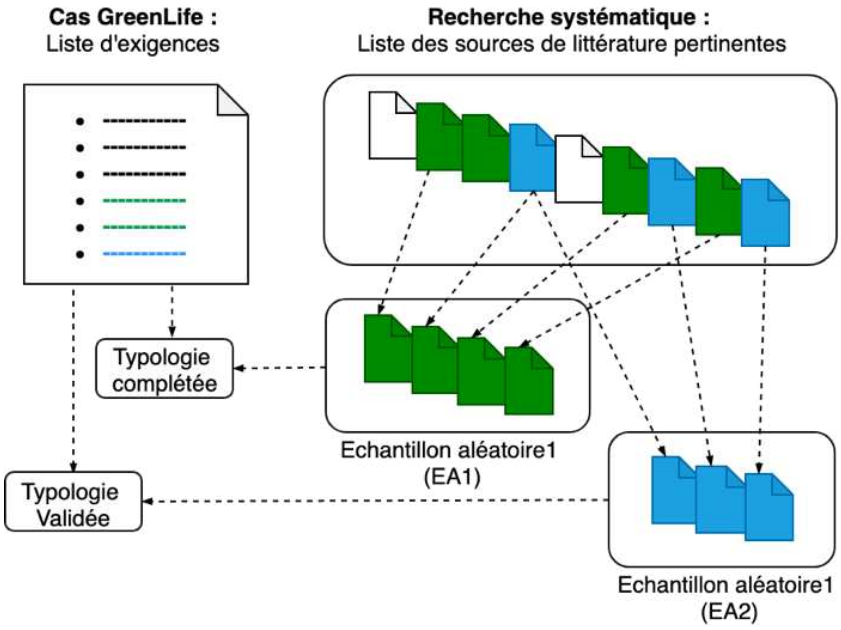

Figure 2-10 : Protocol d'élaboration de la typologie d'exigences

2.4.2. Classification des exigences

Sur la base du cas, et de l'étude de littérature effectuée, nous avons classifié les exigences soulevées en trois catégories : les exigences de domaine qui définissent le segment de marché concerné par les systèmes dérivés, les exigences d'application qui permettent la spécification de systèmes selon les besoins des utilisateurs, et finalement, les exigences d'adaptation qui déterminent les règles de reconfiguration dynamiques des systèmes (Raúl Mazo et al. 2020).

2.4.2.1. Exigences de Domaine

Les exigences de domaine sont les exigences qui sont caractéristiques d'un segment de marché particulier, elles décrivent les constituants organiques, fonctions de base, et autres caractéristiques essentielles que tout système appartenant à ce domaine est susceptible de posséder.

Les fonctions peuvent être qualitatives ou quantitatives et correspondent ainsi à des **exigences fonctionnelles** (Ex_Dom_fun) ou **non-fonctionnelles** (Ex_Dom_NFR). De plus, afin de séparer les fonctions qui doivent nécessairement être accomplies par le système de celles qui ne sont valides que dans des contextes spécifiques, des **exigences de variabilité** (Ex_Dom_Var) sont introduites. Elles présentent d'abord l'ensemble des fonctionnalités, des dispositifs et des constituants pouvant faire partie du système dérivé,

ensuite les dépendances entres ces éléments, leurs domaines de valeurs et finalement, leurs cardinalités (Han et al. 2017) (Abbas et al. 2010) (Muñoz-Fernández et al. 2018). Les valeurs de certaines exigences, appelées **exigences paramétriques** (Ex_Dom_Par), sont déterminées en build-time, et sont donc spécifiées au niveaux du domaine sous forme de paramètres. Les exigences de domaine sont décrites dans la Figure 2-11, et sont rédigées selon la formulation introduite dans le Tableau 2-6.

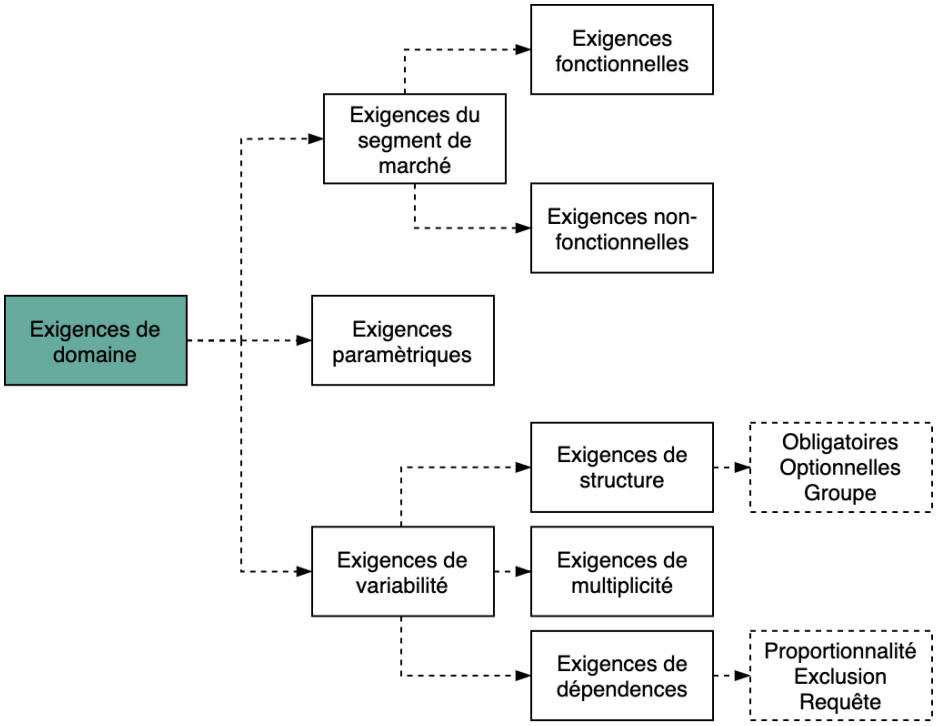

Figure 2-11 : Exigences de domaine

Exemples :

- **Ex_Dom_fun** : Le SdS_Dy doit assurer le monitoring du contexte ;
- **Ex_Dom_NFR** : Le SdS_Dy peut être efficace en termes de consommation électrique ;
- **Ex_Dom_Par** : Quand le niveau du réservoir est supérieur à la valeur optimale (tankLevel > opTankWater), le SdS_Dy doit activer un dispositif d'irrigation
- **Ex_Dom_var** : Le SdS_Dy peut muter entre le Sprinkler, le Dripline, le CenterPivot ou le Rooftop pour assurer l'irrigation ;
- **Ex_Dom_var** : Le SdS_Dy doit utiliser au moins un capteur d'humidité pour capturer l'humidité du champ ;

Type d'exigence	Événement et condition	Action	Artéfact concerné par l'exigences
Ex_Dom_fun	(Si <Condition>) Le SdS_Dy	Doit/Peut *Assurer*	<Tâche>
Ex_Dom_NFR		Doit/Peut *Être*	<Qualité>
Ex_Dom_Var		Doit/Peut *Utiliser*	<Élément>
		Doit/Peut *Composer*	<Élément> avec (au moins <min>/au plus <max>) ? (<ÉlémentY>/<ListeÉléments>)
		Doit/Peut *Combiner/Dissocier*	<ÉlémentX> et (<Élément>/<ListeÉléments>)
		Doit/Peut *Muter Entre*	<ListeÉléments>
		Doit/Peut Utiliser	Au moins <min>/Au plus <max> Instances de < Élément>

Tableau 2-6: Template pour la formulation des exigences de domaine

2.4.2.2. Les exigences d'Application

Les exigences d'application (Ax_App) représentent les exigences exprimées pour un produit dérivé dans le domaine étudié. Elles peuvent être de natures diverses, lesquelles sont présentées dans la Figure 2-12. La formulation des exigences d'application est décrite dans le Tableau 2-7.

- **Les exigences fonctionnelles** (Ex_App_Fun) qui représentent les fonctionnalités retenues du domaine pour le SdS en question (Abbas et al. 2010)
- **Les exigences non-fonctionnelles** (Ex_App_NFR) qui représentent les qualités que le système en question doit assurer (Muñoz-Fernández et al. 2018; Yang et al. 2013; D'Ippolito et al. 2014)
- **Les [exigences de] préférences** (Ex_App_Préf) représentent les choix des utilisateurs finaux. Ces exigences peuvent correspondre à un choix booléen qui exprime la volonté de l'utilisateur à intégrer ou non un artefact de domaine dans son système, comme elles peuvent correspondre à l'instanciation d'un paramètre.
- **Les exigences de proportionnalité** (Ex_ App_Prop) spécifient les règles qui définissent des relations logiques entre divers éléments d'un SdS.
- **Les exigences de coût** (Ex_App_Coût) décrivent les contraintes budgétaires affectées au produit en question.

- **Les exigences d'optimisation** (Ex_App_Opt) qui définissent au moyen d'une fonction objectif un maximum ou un minimum requis pour les valeurs des paramètres (ou des cardinalités) des constituants du SdS (Soares et al. 2017).
- **Les exigences d'autonomie** (Ex_App_Auto) sont spécifiques aux systèmes auto-adaptatifs, et représentent les exigences dérivées d'exigences de plus haut niveau, offrant des alternatives d'adaptation pour les contextes dynamiques (Vassev and Hinchey 2015).

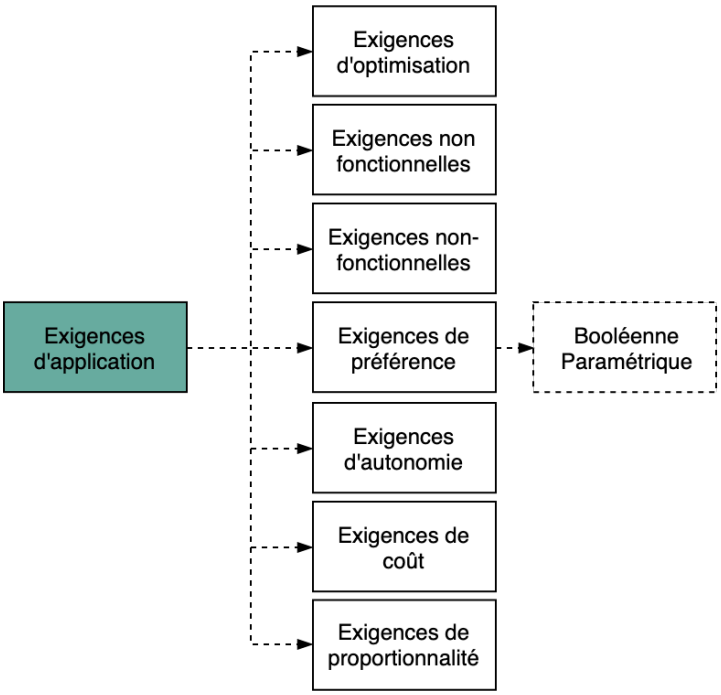

Figure 2-12 : Exigences d'application

Exemples :

- **Ex_App_Fun** : Le SdS_Dy doit utiliser le Mainswater pour dispenser l'eau au champ ;
- **Ex_App_NFR** : Le SdS_Dy doit assurer l'efficacité énergétique ;
- **Ex_App_Préf** : Le SdS_Dy doit utiliser le Dripline avec 114 émetteurs ;
- **Ex_ App_Prop** : Le SdS_Dy doit activer les Dripline avec une pression respective de 15,30 et 40 psi pour des nombres d'émetteurs égale à 114, 200 et 232 ;
- **Ex_App_Coût** : Le SdS_Dy ne doit pas dépasser le budget de 2000€ ;
- **Ex_App_Opt** : Le SdS_Dy doit optimiser le nombre de capteurs d'humidité actifs ;
- **Ex_App_Auto** : Le SdS_Dy peut muter entre les ampoules et le AC pour assurer le control de la température du champ.

Type d'exigence	Événement et condition	Action	Artéfact concerné par l'exigences
Ex_App_Fun	(Quand <événement>), (si <condition>), < le SdS_Dy >	Doit *Assurer*	<Tâche>
Ex_App_NFR		Doit *Être*	<Qualité>
Ex_App_Préf		(Ne) Doit (pas) *Utiliser*	<Element> (avec la valeur <Paramètre>)
Ex_ App_Prop		DOIT *sélectionner respectivement*	(ListeParams) <ElémentX>, si <Élément> (ListeParams)
Ex_App_Coût		Doit *coûter*	Au moins <minCoût> / Au plus <maxCoût>
Ex_App_Opt		DOIT *minimiser/maximiser*	(la valeur de/ les instances de) <Élément>
Ex_App_Auto		Doit/Peut *Utiliser*	<Elément>
		Doit/Peut *Composer*	<Élément> avec (au moins <min>/au plus <max>) ? (<ÉlémentY>/<ListeEléments>)
		Doit/Peut *Combiner/Dissocier*	<ElémentX> et (<Élément>/<ListeEléments>)
		Doit/Peut *Muter Entre*	<ListeEléments>

Tableau 2-7: Template pour la formulation des exigences d'application

2.4.2.3. Exigences d'Adaptation

Les exigences d'adaptation sont des exigences qui décrivent le comportement du système dans un contexte dynamique, et présentent les éventuelles adaptations nécessaires pour garantir la satisfaction des fonctionnalités exigées ou espérées du système auto-adaptatif. Ainsi, plusieurs types d'exigences peuvent être exprimés (dos Santos et al. 2019) (Duarte et al. 2018) (Muñoz-Fernández et al. 2018) (Dar et al. 2018). Les exigences d'adaptation sont décrites dans la Figure 2-13. Elles sont formulées selon le gabaris d'exigences présenté dans le Tableau 2-8.

- **Les exigences temporelles** (Ex_Adp_Temp) déterminent le temps, l'ordre ou la fréquence avec lesquels des exigences doivent être satisfaites.
- **Les exigences Relaxables** (Ex_Adp_Rlx) font référence à des exigences qui sont nécessaires dans des contextes particuliers, mais qui peuvent s'avérer moins

indispensables dans le cadre de changements de l'environnement ou du comportement des utilisateurs (Whittle et al. 2010).

- **Les exigences AwRqs** (Ex_Adp_Aw) sont des exigences de sensibilité qui contraignent les degrés de succès ou d'échec de la mise en œuvre d'autres exigences d'adaptation (Souza et al. 2013)
- **Les exigences d'optimisation** (Ex_Adp_Opt) maintiennent leur rôle de maximisation ou de minimisation de valeurs de paramètres ou de cardinalités des constituants du système après l'exécution de ce dernier (Uthariaraj and Florence 2011).
- **Les exigences contextuelles** (Ex_Adp_Ctx) les exigences qui sont valables dans des contextes particuliers (Mori 2012)
- **Les exigences de résilience (aussi appelées exigences d'évolution EvRqs)** (Ex_Adp_Ev) déterminent les exigences qui spécifient la réponse à apporter en cas d'échec de la mise en œuvre d'autres exigences d'adaptation (Souza et al. 2012)

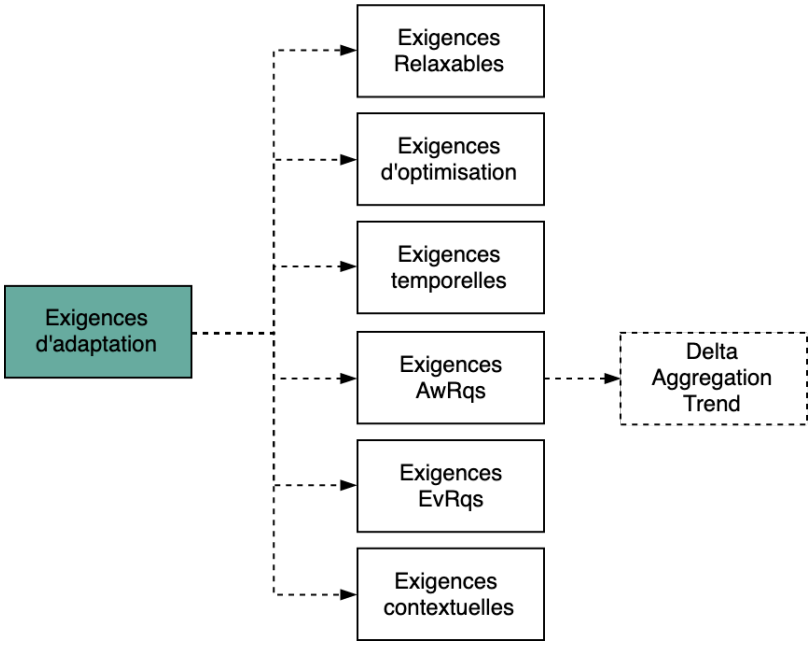

Figure 2-13 : Exigences d'adaptation

Exemples :

- **Ex_Adp_Temp** : Le 1er octobre, le SdS_Dy doit activer le mode Automne/Hiver
- **Ex_Adp_Rlx** : Le SdS_Dy doit utiliser autant de capteur d'humidité que possible, éventuellement, tous les niveaux de batterie doivent être supérieurs à 40%
- **Ex_Adp_Aw** : Le taux d'échec de l'exigence E_P/É_14 ne doit pas dépasser 4 fois.

- **Ex_Adp_Opt** : Quand (PowerFailure=True), le SdS_Dy doit minimiser le nombre de Master
- **Ex_Adp_Ctx** : Si le (nombre de Humidity_Sensors dont le niveau de batterie > 60 est < 4), le SdS_Dy doit maintenir le niveau de batterie des Humidity_Sensor en mode Master > 40%
- **Ex_Adp_Ev** : Quand l'exigences (Ex_Adp_Aw) n'est pas satisfaite, le SdS_Dy doit satisfaire l'exigence E_P/É_14

<table>
<tr><th>Type d'exigence</th><th>Événement et condition</th><th>Action</th><th>Artéfact concerné par l'exigences</th></tr>
<tr><td rowspan="4">Ex_Adp_Temp</td><td>À<temps/date></td><td colspan="2"><Exigence></td></tr>
<tr><td>Pendent <durée></td><td colspan="2"><Exigence></td></tr>
<tr><td>Entre (temps/date) Et (temps/date)</td><td colspan="2"><Exigence></td></tr>
<tr><td>(Avant/après) (temps/date)</td><td colspan="2"><Exigence></td></tr>
<tr><td>Ex_Adp_Rlx</td><td rowspan="8">(Quand <événement>), (si <condition>), <le SdS_Dy></td><td>Doit <Optimisation>, éventuellement,</td><td>< Élément > doit être (égale à <valeur>/optimisé)</td></tr>
<tr><td rowspan="2">Ex_Adp_Aw</td><td>Le taux d'échec/succès de</td><td><Exigence> (<durée>)ne doit pas dépasser <optimum></td></tr>
<tr><td>L'<Élément> doit</td><td>être instancié dans la limite de <Durée></td></tr>
<tr><td>Ex_Adp_Opt</td><td>Doit minimiser/maximiser</td><td>(la valeur de/ les instances de) <Élément></td></tr>
<tr><td rowspan="2">Ex_Adp_Ctx</td><td>Doit augmenter /diminuer /changer /modifier</td><td><Élément></td></tr>
<tr><td>Doit maintenir</td><td><ÉlémentX> <opération> (<Élément> /<Valeur>)</td></tr>
<tr><td>Ex_Adp_Ev</td><td>Doit augmenter /diminuer /changer /modifier</td><td><Exigence></td></tr>
</table>

Tableau 2-8: Template pour la formulation des exigences d'adaptation

2.5. Conclusion

Le cas présenté dans ce chapitre détaille les exigences relatives à la solution GreenLife dont l'objectif est d'automatiser le processus d'irrigation d'un champ d'agriculture, et ce en répondant aux besoins d'une multitude de clients, dont les besoins s'adaptent et évoluent. Les exigences réutilisables de domaine relatifs à la solution GreenLife d'un côté, et les exigences d'application et d'adaptation de Maria d'un autre, révèlent le besoin de

spécifier une multitude extrêmement diversifiée d'exigences de configuration, notamment des exigences fonctionnelles, non fonctionnelles, structurelles, opérationnelles, paramétriques, de composition, de préférence, de proportionnalité, d'optimisation, de temps, de coût, d'autonomie, relaxables, d'optimisation, contextuelles, de sensibilité et d'évolution.

Bien que cette typologie des exigences s'appuie sur la typologie standard des exigences [2] (de produit, de processus, fonctionnelles, non fonctionnelles, qualité, contrainte) (Lin et al. 1996) (IEEE 2009), elles se différencient de ces dernières par le fait qu'elles ne soient pas toujours obligeantes. En réalité, elles sont vérifiables dans des contextes spécifiques, et ne le sont pas dans d'autres. La spécification de ces exigences, que l'on peut qualifier de « dynamiques », permet la spécification de SdS_Dy hautement reconfigurables, répondant à des besoins différents, tout en assurant l'évolution intrinsèque de cette catégorie de systèmes.

[2] https://www.ireb.org/content/downloads/2-syllabus-foundation-level/ireb_cpre_syllabus_fl_fr_v221.pdf

Chapitre 3 État de l'art des modèles de spécification de systèmes de systèmes dynamiques

3.1. Introduction

Il n'existe pas de méthode, de langage, ou de framework parfait. Ce qui a du sens, c'est une méthode, un langage, ou un framework, qui soit le plus approprié pour résoudre un problème spécifique, ou qui réponde à un besoin stratégique d'une catégorie de systèmes ou d'organismes (Shenhar 2001). Avant de proposer le langage qui permette la spécification des exigences de SdS_Dy, une revue de la littérature s'avère nécessaire pour étudier comment d'autres langages sont conçus et analyser ces langages par rapport aux caractéristiques des SdS_Dy. Ainsi, les questions de recherche abordées dans ce chapitre, sont définis comme suit :

- **QR_L1 :** La notation prend-elle en compte les exigences de domaine des SdS_Dy, à partir desquels plusieurs SdS individuels peuvent être générés?
- **QR_L2 :** La notation permet-elle de spécifier la typologie déduite suite à l'étude de cas du chapitre 2 ?
- **QR_L3 :** La notation permet-elle de spécifier des exigences dynamiques ?

Ces questions de recherche s'ajoutent aux questions de recherche principales présentées dans le Chapitre 1 , et elles portent le préfixe L, pour préciser qu'il s'agit de questions de recherche de littérature.

Avant de répondre à ces questions, le chapitre introduit quelques approches reconnues pour la spécification de systèmes dynamiques ou de systèmes réactifs, et analyse les notations qu'elles proposent afin de juger de leur utilité pour la spécification des SdS_Dy.

3.2. Aperçu des approches de spécification de comportement dynamique

Qu'il s'agisse des sous exigences de domaine, d'application ou d'adaptation, cette thèse s'intéresse à la spécification de cette typologie d'exigences, dans une perspective de (re)configuration. C'est à dire qu'il s'agit de spécifier correctement les exigences des SdS_Dy, retenues pour un contexte donné, afin de générer une configuration valide dans ce même contexte. Lorsque la configuration n'est plus valide, ou lorsque le contexte change, de nouvelles exigences doivent être retenues.

Dans le cadre de cette thèse, la « configuration » fait référence à un ensemble de fichiers qui constituent un produit logiciel valide par rapport aux exigences qui le spécifient (Estublier 2000). Ainsi, la démarche de configuration consiste à :

1. identifier et définir les rôles de chaque élément constituant le SdS_Dy,
2. identifier, collecter et surveiller les données du contexte,
3. contrôler le système, du composant le plus élémentaire (sous-systèmes du SdS_Dy), au système complet (le SdS_Dy dans sa globalité), en choisissant parmi l'ensemble des fichiers de configurations possibles pour chaque système, celui qui soit valide dans le contexte courant.

La reconfiguration consiste à ajuster la composition du SdS_Dy, ainsi que son comportement, afin d'assurer la réalisation des exigences valides dans un contexte donné, à l'issue d'un changement (Macías-Escrivá et al. 2013). De manière simplifiée, la configuration d'un système peut être présentée comme une description des composants actifs du système, ainsi que des paramètres et procédures de fonctionnement de ces composants. La première action fait référence à la structure du système, i.e. la nature de ses éléments, de leur occurrence, ainsi que des relations qui les unissent. La deuxième action fait référence au comportement du composant, i.e. les fonctionnalités qu'il a à offrir. Cette opération de configuration est possible grâce à une variété de méthodes et d'arsenaux méthodologiques existants pour guider dans la spécification de systèmes réactifs. La plupart d'entre elles sont axées sur la notion de buts (Goal Oriented Requirements Engineering : GORE en anglais). La principale raison est la capacité de ces approches à modéliser le système et à modéliser son environnement (van Lamsweerde 2001). La deuxième tendance est l'ingénierie système, qui est au cœur du développement de Système de Systèmes (System of System en anglais). Les approches de spécification de SdS (Model-Based System Engineering : MBSE en anglais) tels que SysML, sont dès lors des modèles opportuns pour la spécification d'exigences de plus haut niveau (SdS_Dy), jusqu'aux exigences opérationnelles de bas niveaux (toute la hiérarchie du SdS_Dy). La troisième approche est l'ingénierie des lignes de produits logiciel dynamiques. Elle permet la spécification de produits multiples, partageant des caractéristiques similaires, tout en étant différents les uns des autres. Ce paradigme propose une ébauche de solution aux problèmes liés à la spécification des systèmes auto-adaptatifs, en considérant une configuration comme un élément réutilisable, et en dérivant pour certains changements du contexte, une nouvelle configuration du système ou du SdS_Dy (Muñoz-Fernández et

al. 2014; Bencomo et al. 2008; Pfannemueller et al. 2017; Hallsteinsen et al. 2008; Capilla et al. 2014).

3.2.1. Modélisation selon l'approche par buts

Un modèle de buts est un modèle centré sur les objectifs exprimés pour un système, afin de spécifier les exigences qui en découlent, tout en prenant en compte les corrélations existantes avec l'environnement dans lequel il est déployé (Rolland 2003). Plusieurs variations des modèles de buts existent dans la littérature en Ingénierie des Exigences, notamment, les modèles i* (Yu 1997), KAOS (Dardenne et al. 1993), Tropos (Bresciani et al. 2004), GLR (ITU-T 2008), etc. Notre étude comparative prend en compte l'ensemble des modèles de buts, qualifiés dans les revues de littérature telles que (Granjal et al. 2014) et (Darwish and Zohdy 2016), de méthodes « matures » de spécification des exigences. Certains travaux qui étendent les modèles de buts avec des concepts en rapport avec la gestion de la dynamique sont ignorés ici. En effet, bien que ces derniers répondent à des limitations qui nous intéressent, ils ne sont pas matures et donc pas intéressants pour souligner notre propos. Par exemple, les travaux de (Semmak et al. 2010) et (Tueno et al. 2017) qui étendent les modèles KAOS avec les dépendances logiques et temporelles, ou encore les travaux de Muñoz et al. (Muñoz-Fernández et al. 2018) et Alférez et al. (Alférez et al. 2014) qui développent les dépendances entres exigences, pour prendre en compte des aspects liés à la variabilité sont tout à fait intéressants, mais nous les considèrerons dans un deuxième temps.

3.2.1.1. Aperçu des concepts des modèles selon l'approche par buts

Les principaux concepts nécessaires pour la spécification des exigences, à partir des modèles de buts, sont les suivants :

- **Le but :** Une propriété attendue du système et exprimée par une partie prenante (ingénieur du système, client, utilisateur, etc). La compréhension de certains buts nécessite leur raffinement en sous-buts qui rendent plus explicite la spécification du système. Les modèles des buts ont en général une structure d'arbre. Ainsi, un but peut être une racine, un sous-but, ou un but feuille, et peut être relié à une exigence, une supposition ou à une propriété de domaine. Un but peut faire référence à un comportement qui doit être réalisé à l'aide du système (Hard Goal en anglais). Mais il

peut aussi faire référence à une qualité qui doit être garantie et maintenue à un certain degré (Soft Goal en anglais).

- **L'exigence** : définie dans KAOS comme une sous-catégorie de but de bas niveau, l'exigence représente la mise en œuvre opérationnelle qui doit être directement réalisée par le système ou par l'un de ses constituants.
- **L'hypothèse:** Une sous-catégorie de buts de bas niveau supposée réalisée par l'environnement du système.
- **L'agent :** L'entité responsable de l'accomplissement d'un but exprimé. Il peut faire référence à un humain, le système à construire, un composant du système à construire, ou un autre système sociotechnique.
- **L'opération :** Décrit le comportement qu'un agent doit ou peut réaliser.

Un système peut être modélisé en respectant le principe de séparation des préoccupations au moyen de ces quelques concepts. Ainsi, un modèle de buts, un modèle de responsabilités, un modèle d'opérations et un modèle objet peuvent représenter, respectivement, les exigences attendues du système, les entités responsables de leur accomplissement, les opérations nécessaires à cet effet, ainsi qu'une vue de la structure globale du système.

3.2.1.2. Modélisation du cas GreenLife : Modèles selon l'approche par buts

Afin de mieux cerner l'utilité des modèles de buts et d'éprouver leur pouvoir d'expression pour la spécification de systèmes de systèmes dynamiques, nous modélisons à l'aide de modèles de buts les exigences spécifiées pour la solution GreenLife en général, et pour la cliente Maria spécifiquement. La première étape dans ce processus de modélisation revient à définir les principaux buts du SdS_Dy d'irrigation, représentés dans la Figure 3-1, afin ensuite de les raffiner du plus haut niveau, jusqu'aux exigences.

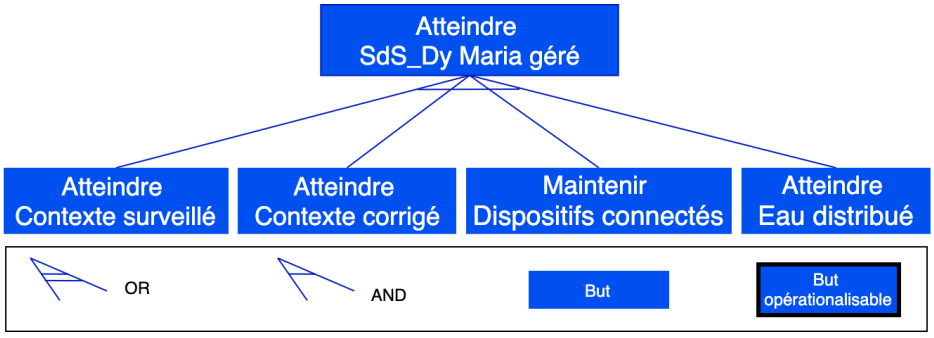

Figure 3-1: Cas Maria – Les buts selon l'approche KAOS

Comme il est décrit dans la Figure 3-1, chaque but est raffiné en sous buts lesquels sont à leur tour raffinés en exigences qui peuvent elles-mêmes être décomposées. En effet, l'exigence globale relative à la gestion du SdS_Dy de Maria revient à surveiller le contexte et le contrôler en conséquence, à connecter les dispositifs nécessaires à cet effet, et finalement à distribuer l'eau.

La Figure 3-2 présente le raffinement du but « Contexte surveillé » qui est réalisé par la satisfaction des quatre exigences qui en découlent.

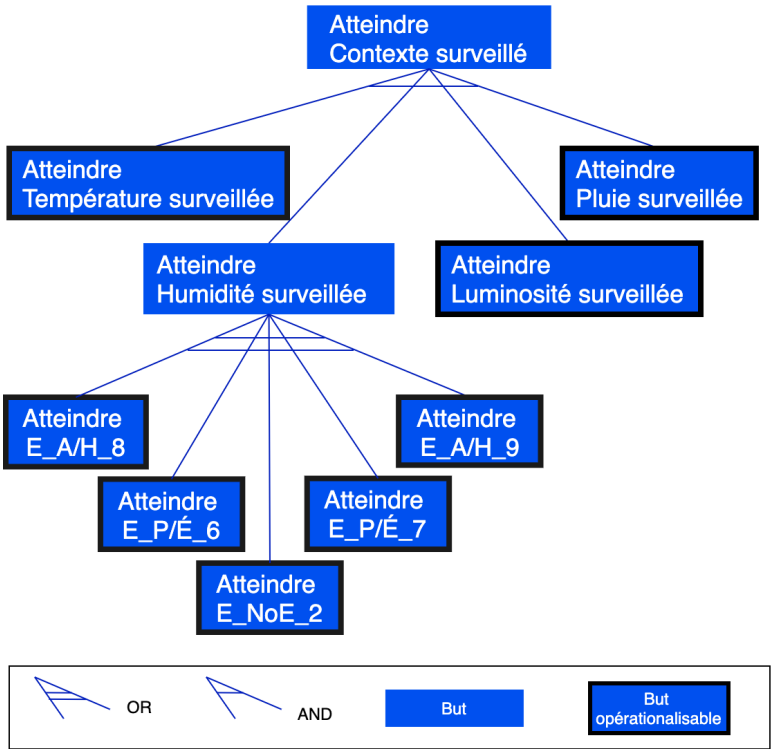

Figure 3-2 : Cas Maria – Les exigences spécifiées selon l'approche KAOS

Ces exigences sont toutes de bas niveau, mis à part le but « Humidité surveillée », qui peut à son tour être décomposé en 5 sous-exigences à satisfaire dans des contextes différents, et donc alternatives comme le montre le raffinement OR. E_A/H_8, E_P/E_6, E_NoE_E, etc., sont des identifiants d'exigences dont la description complète est disponible dans les tableaux 2-2, 2-3 et 2-4.

Chaque exigence est connectée d'un côté aux agents qui sont responsables de sa satisfaction et d'un autre côté aux différentes opérationnalisations qui sont déclenchées lorsqu'un évènement se produit. L'exigence « Sprinkler activé » est réalisée par la mise en œuvre de l'une des exigences « E_A/H_1 », « E_P/E_1 » ou « E_ P/E _2 », via les opérations

« AW Rotor Mode », « Spray Mode » ou « SS Rotor Mode ». L'exigence mère est sous la responsabilité de l'élément Sprinkler. Les connections entre exigences, opérationnalisations et agents sont illustrées dans la Figure 3-3.

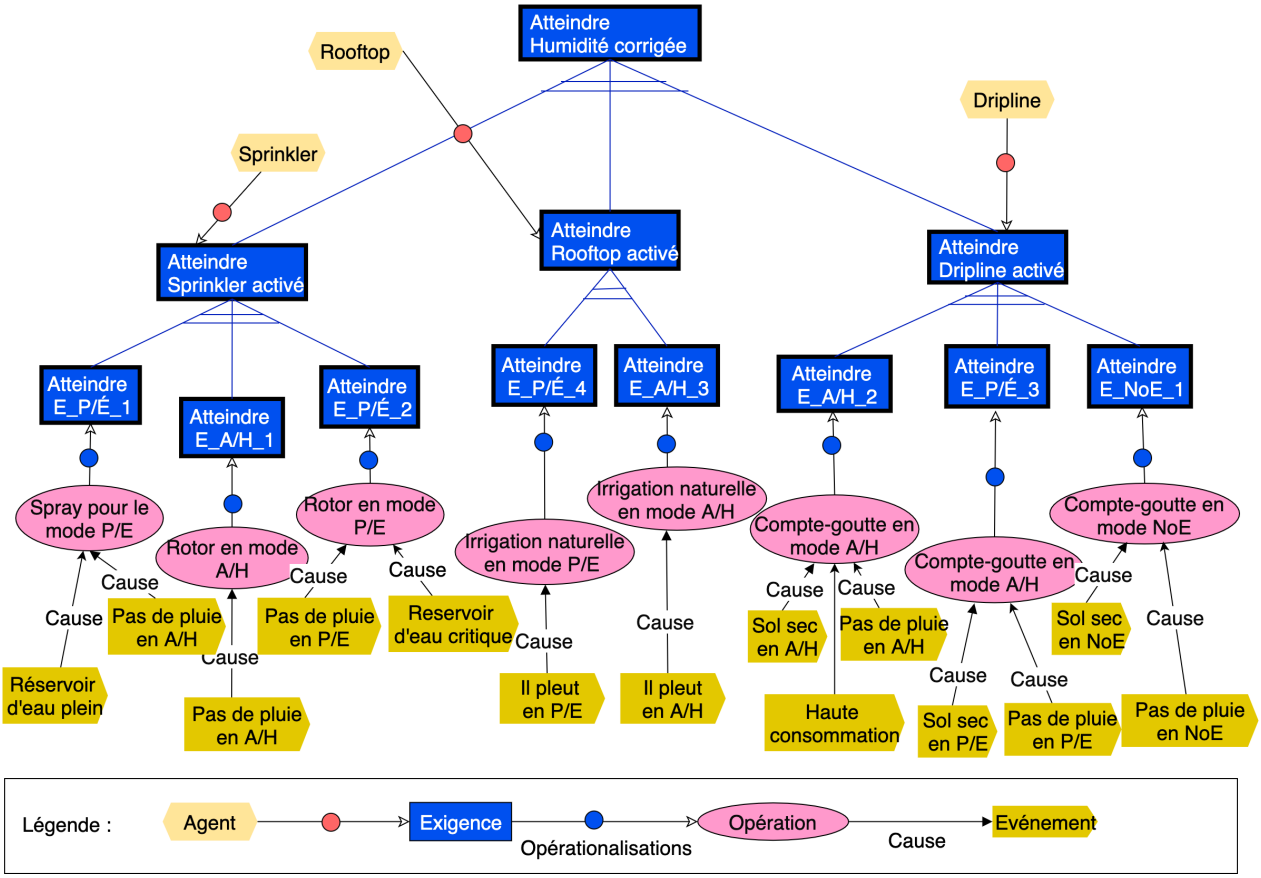

Figure 3-3 : Cas Maria – Les opérations et agents selon l'approche KAOS

La spécification de SdS_Dy selon l'approche par but permet en effet de définir un ensemble de buts qui peuvent être atteints de plusieurs manières, en permettant leur raffinement en exigences opérationnalisables distinctes. Le choix d'une opérationnalisation (et donc d'une exigence à satisfaire) au détriment d'une autre est guidé par l'état du contexte du système, décris par le biais d'événements. Cette notation demeure cependant limitante. Des exigences telles que « E_P/É_7 : Si le niveau de batterie d'au moins 1 capteur d'humidité est inférieur à 60% (BL[1<60 \/... BL[7] <60) les 3 capteurs d'humidité dont les niveaux de batteries sont les plus élevées sont en mode Master » ne peuvent pas facilement être formalisées avec KAOS. Les muti-instances ne sont pas considérées par la notation, les états de changement de configuration ne sont pas spécifiés et la description de l'environnement nécessaire pour déclencher ces changements est limitées.

3.2.2. Modélisation selon l'approche SySML

SySML est une variante du langage de modélisation UML, spécialement conçue pour modéliser des systèmes dans le cadre de l'ingénierie de systèmes (MBSE) (Delligatti 2014). Tout comme UML, les modèles SysML sont standardisés par l'OMG (OMG 2017) et sont particulièrement pensés pour la spécification explicite des exigences, des structures physiques de systèmes et de leur comportement fonctionnel, au moyens de différents modèles intervenant dans chacune de ces trois perspectives.

- **Un modèle d'exigences** est une représentation graphique des exigences des parties prenantes du système. Ce modèle représente les exigences sur plusieurs niveaux d'abstraction et définit les types de relations entre celles-ci. Le modèle d'exigences assure une forme de « traçabilité » en permettant de tirer des dépendances entre les exigences et les éléments d'autres modèles, qui les satisfont, les vérifient ou les raffinent.
- **Les modèles de structure** spécifient ce qu'est le système sous forme de blocs organisés pour décrire la hiérarchie du système et de ses constituants. Le diagramme de définition de blocs internes décrit la structure interne d'un système en termes de composants, de ports et de connecteurs. Le diagramme de package est utilisé pour organiser les modèles. Finalement, le diagramme paramétrique représente les constantes sur les valeurs des propriétés du système, telles que les performances ou encore la fiabilité.
- **Les modèles de comportement** spécifient ce que le système doit faire, en conséquence de l'état et la nature de son environnement. Le modèle de cas d'utilisation fournit une description détaillée des fonctionnalités que le système offre, ainsi que les acteurs qui les invoquent ou qui participent à leur réalisation. Le modèle d'activité représente le flux de données et l'enchaînement des activités. Le modèle de séquences représente l'interaction entre les parties collaboratrices d'un système. Et finalement, le diagramme de machine à états décrit les transitions d'états et les actions qu'un SdS effectue en réponse à des événements.

Dans le reste de cette partie, les concepts de trois diagrammes de l'approche SysML seront détaillés. D'abord le diagramme d'exigences, parce qu'il définit les exigences et assure leur traçabilité sur tout le processus de modélisation, ensuite, le diagramme de définition de

blocs, parce qu'il décrit la composition du SdS_Dy et finalement, le diagramme de machine à états finis, parce qu'il peut être utilisé comme système de contrôle, où le comportement exigé est stocké dans un état et les transitions jouent le rôle de médiateur, entre l'environnement et le système, afin de ramener ce dernier dans un état convenable.

3.2.2.1. Aperçu des concepts des modèles selon l'approche SysML

SysML est un arsenal méthodologique qui implique plus d'une cinquantaine de concepts. Dans ce qui suit sont discutés ceux qui interviennent dans les modèles sélectionnés dans la section précédente (Delligatti 2014).

- **L'exigence** spécifie une capacité ou une condition qui doit être satisfaite. Une exigence peut spécifier une fonction que le système doit exécuter ou une condition de performance qu'un système doit atteindre. Les exigences peuvent être composées en sous exigences, dérivées d'autres exigences, comme elles peuvent être raffinées ou satisfaites par des éléments de modèles autres que le modèle d'exigence.
- **Le bloc** est une unité basique dans la modélisation de système avec SysML. Il fait référence du système lui-même, à un sous-système, ou à un composant matériel ou logiciel. Les blocs sont primordiaux pour la compréhension d'un système tout comme les relations qui lient ses sous-composants. Ces relations peuvent exprimer une généralisation, une association ou une dépendance entre au moins deux blocs.
- **L'état** est une entité qui décrit le comportement d'un système, face à une action interne ou externe à celui-ci, ou en réaction à un événement temporel. Ce concept offre à SysML le pouvoir d'exprimer le comportement de différents niveaux d'abstraction du système. Ainsi, un état spécifie un comportement du système, en définissant les différents états des sous-systèmes qui le composent.
- **La transition** représente le changement entre un état source et un état destination. Celle-ci est guidée par une expression conditionnelle dont la satisfaction incarne la manifestation d'un événement dans l'environnement du système.

3.2.2.2. Modélisation du cas GreenLife : Modèles selon l'approche SysML

Les exigences de la solution GreenLife et du produit Maria peuvent être représentées par un digramme d'exigences. Par conséquent, les principales fonctionnalités attendues du système, à savoir le monitoring du contexte, le contrôle du contexte, la gestion de la

communication, et le contrôle d'eau, sont représentés dans un diagramme d'exigences, tel qu'il est illustré dans la Figure 3-4 (a). L'exigence 3 par exemple est une exigence de haut niveau. Elle est donc affinée afin de spécifier clairement les exigences relatives au contrôle de température, d'humidité et de luminosité. Le détail de cette exigence est illustré dans la Figure 3-4 (b).

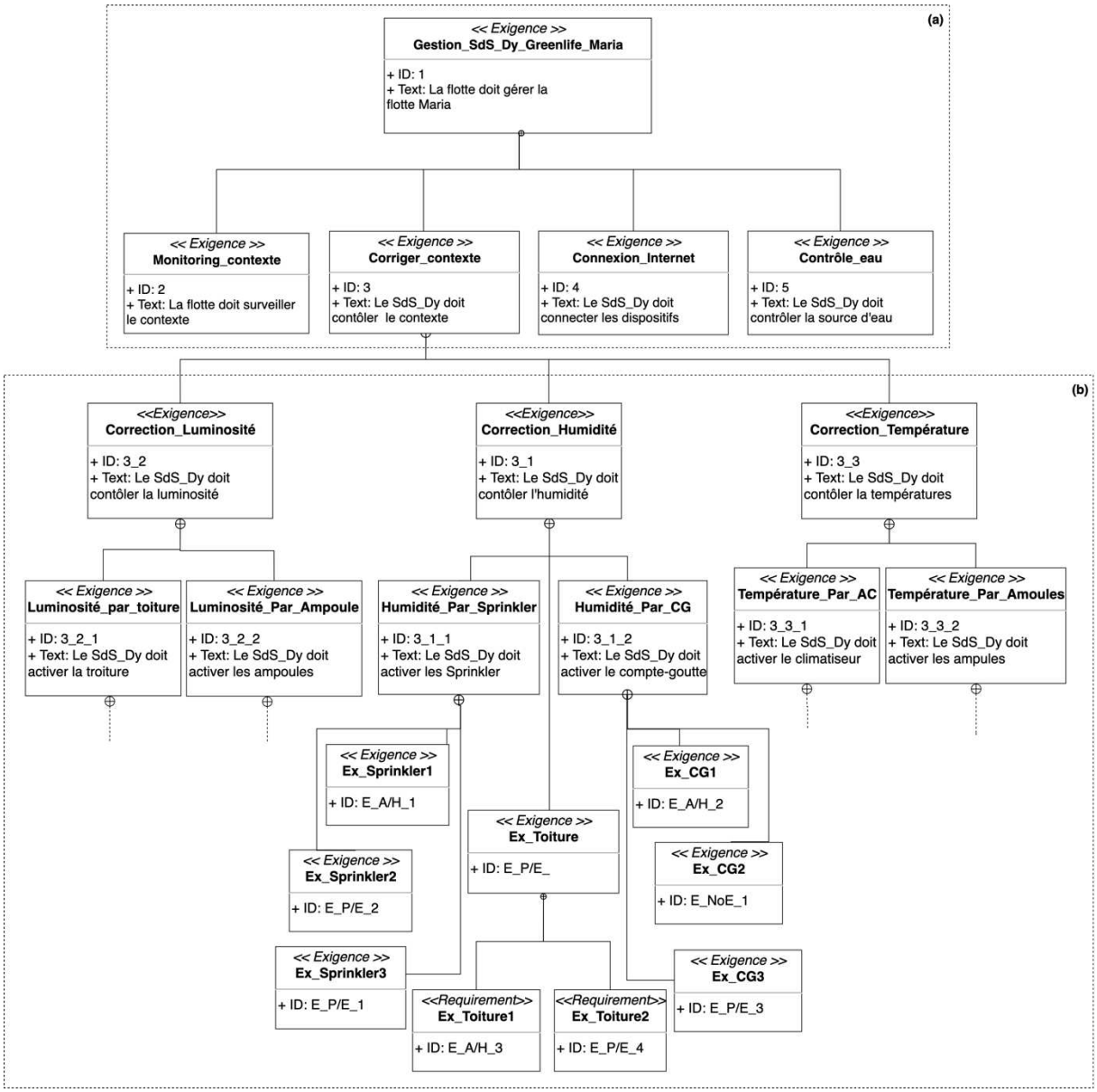

Figure 3-4: Cas Maria – Modèle d'exigences selon l'approche SysML[3]

[3] Les textes des exigences liées au contrôle d'humidité sont disponibles dans les tableaux 2-2, 2-3 et 2-4

La structure de la solution GreenLife et du produit Maria peuvent être représentés par un diagramme de blocs. Les sous-systèmes constituants le système d'irrigation sont des capteurs, des actuateurs, des systèmes de connexion et des systèmes de distribution d'eau. Chaque sous système, ainsi que sa hiérarchie, sont représentés par des blocs connectés par des associations traduisant la nature de leur liaison, tel qu'illustré dans la Figure 3-5.

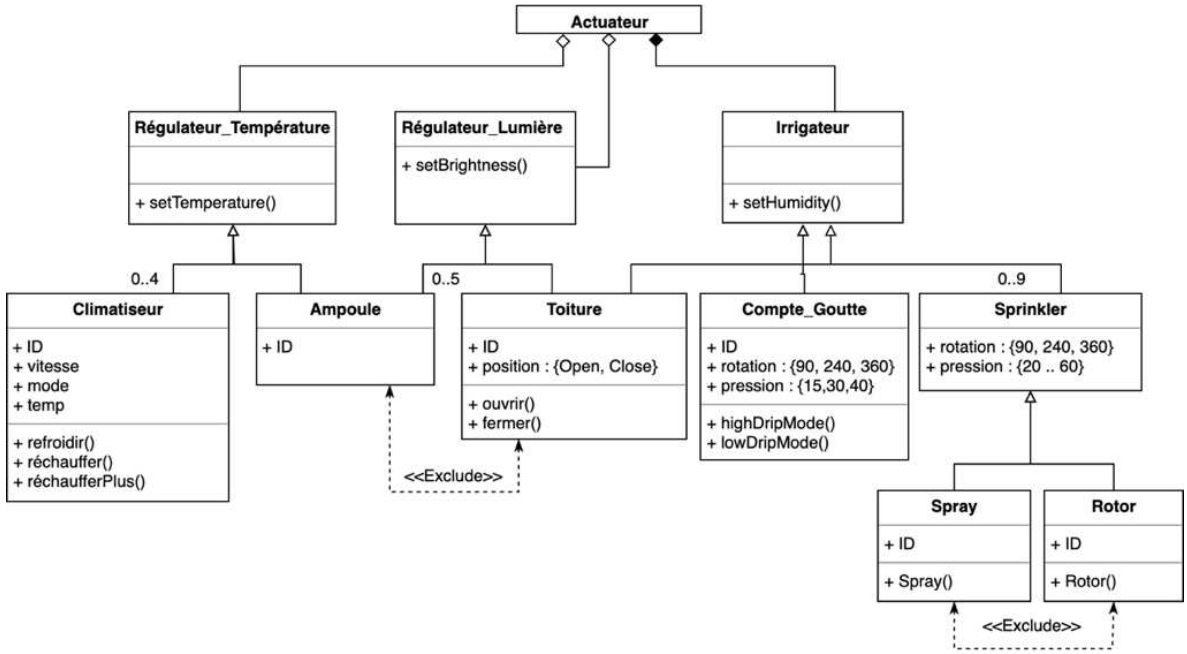

Figure 3-5: Cas Maria – Dépendances du bloc « Controller » selon l'approche SysML

Le bloc « *actuateur* » comprend les dispositifs d'irrigation ainsi que les régulateurs de luminosité et de température. L'association de composition entre le bloc « *actuateur* » et le bloc « irrigateur » traduit bien la présence obligatoire de cet élément dans le système. Cependant, les autres dispositifs, étant optionnels, sont connectés au bloc « *actuateur* » via une agrégation, en guise de leur nature éphémère dans le SdS. Le comportement dynamique de chacun des dispositifs impliqués dans le contrôle de l'environnement est décrit dans un modèle de machine à états finis. La Figure 3-6 présente une partie du modèle global.

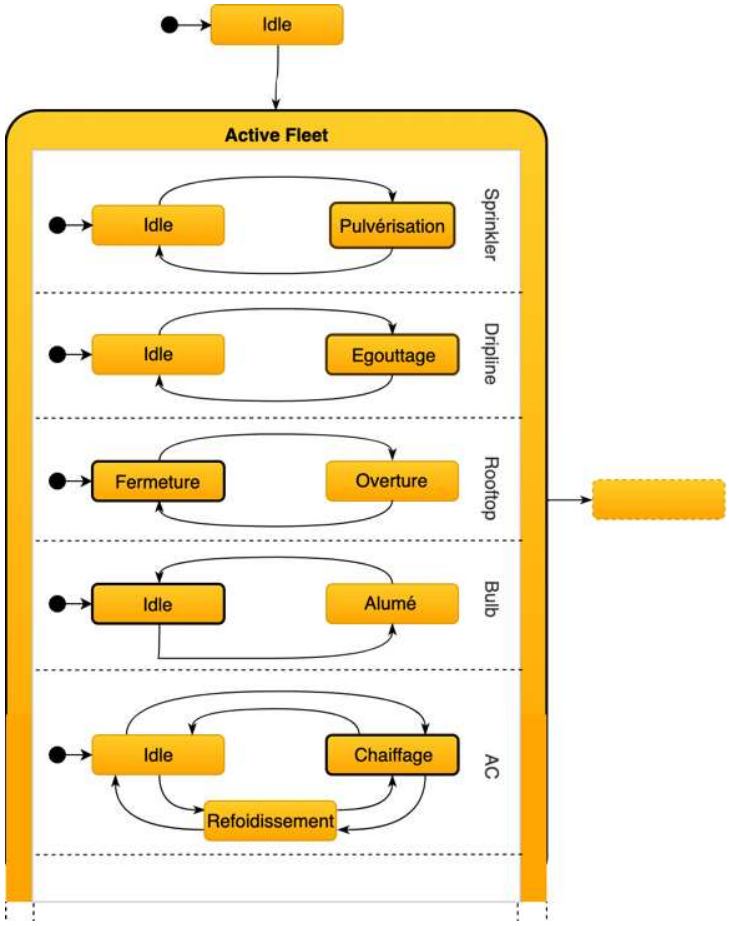

Figure 3-6 : Cas Maria Partie du modèle machine à états selon l'approche SysML

Le graphe machine à états définit le comportement d'un système en termes d'états et de transitions portant le système d'un état source à un état destination. L'activation de tout état engendre l'activation de toutes les actions connectées à celui-ci. Ainsi, suite à la production d'un événement tel que «SoilHumidity < NormalHumidity », l'état du Sprinkler, à titre d'exemple, passe de Idle à Spraying. Cette transition active l'action /LunchSpray().

Les modèles SySML répondent à des besoins de spécification caractéristiques aux systèmes de systèmes. Le principal avantage de cette notation est son pouvoir à convertir les atouts d'une notation typiquement centrée sur l'objet, sur un système en entier. Ce changement de perspective a particulièrement renforcé le pouvoir d'expression du comportement, notamment, par le biais de modèles tels que les de machine à états, ceux-ci étant désormais capables de décrire le comportement d'une entité quelle que soit sa hiérarchie, depuis le composant logiciel jusqu'au système de systèmes. Néanmoins, ces modèle se limitent à la spécification des transitions de fonctions et non à celle des configurations. Par ailleurs, bien que les modèles SysML soient plus expressifs que les

modèles de buts dans la spécification de la structure, étant capables de définir des multi-instances et des dépendances plus élaborées, ils demeurent insuffisants et limitants dans le cas des exigences de systèmes de systèmes dynamiques, telles que « E_P/É_2 : Quand le niveau du réservoir est entre la valeur optimale et minimale (MinTankWater < tankLevel < opTankWater), le SdS_Dy doit activer les sprinkler 1 à 4 en mode Rotor, avec une pression 40psi, et une rotation de 240° ».

3.2.3. Modèle selon l'approche des LdPD

L'ingénierie des lignes de produits logiciels (LdP) (Software Product Lines : SPL en anglais) consiste à concevoir et à créer un ensemble d'éléments partageant des exigences, des fonctionnalités ou des architectures communes, tout en répondant aux besoins d'une catégorie spécifique de parties prenantes (Pohl et al. 2005). Les chercheurs et les ingénieurs s'intéressent à ce paradigme pour résoudre les difficultés liées à la conception et développement des familles de systèmes logiciels, et ce, en définissant les éléments réutilisables dans le cadre de l'ingénierie de domaine, et à en faire usage pour la dérivation de produits spécifiques, et ce dans le cadre de l'ingénierie d'application. Les variations existantes entre des systèmes appartenant à une même ligne de produits représentent la variabilité.

La dérivation d'un produit, dans le cadre des lignes de produits logiciels, est une activité séparée de la conception. Dans ce dernier cas, on parle d'ingénierie de domaine, alors que dans le premier, on parle d'ingénierie d'application. Une fois la dérivation réalisée, le produit peut être testé, et exécuté. La modélisation du domaine, et la dérivation d'applications, constitue l'ingénierie des lignes de produits (ILdP). Plus d'une cinquantaine de langages de modélisation des LDP ont été proposés au fil des années. Six d'entre eux ont été étudiés et analysés par Achtaich et al. dans (Achtaich et al. 2016) selon une variante du cadre de comparaison proposé par (Djebbi and Salinesi 2006). Ces langages descendent de l'approche FODA (Kang et al. 1990), et ont évolué vers des notions dérivées d'approches telles que UML (Eriksson et al. 2005; Streitferdt et al. 2005; Clauß and Jena 2001), RSEB (Favaro and Mazzini 2009) ou encore OVM combinée à SYSML (Dumitrescu 2014). La reconfiguration des SdS_Dy, cependant, a souvent lieu pendant que le SdS est en cours d'exécution. Par conséquent, l'ingénierie des lignes de produits logiciels dynamiques (Dynamic Software Product Lines : DSPL en anglais), devient un choix plus pertinent. Car il maintient les capacités d'ingénierie des LdP même après

l'exécution. Parmi les notations des LdPD, il convient de citer celles proposées par (Cetina et al. 2009), (Morin et al. 2008) et (Sawyer et al. 2012b).
Dans ce qui suit sont présentés les modèles intervenant dans la spécification d'un SdS_Dy, selon l'approche proposée par Sawyer et. al (Sawyer et al. 2012b). Ce choix est basé sur la multitude et la clarté des documents qui détaillent ou utilisent cette notation, notamment dans (Sawyer et al. 2012a; Dounas et al. 2015; Muñoz-Fernández et al. 2014). En plus, cette notation, étant inspirée des modèles de buts, complète les modèles préalablement établis, dans la Section 3.2.1.

3.2.3.1. Aperçu des concepts des modèles selon l'approche des LdPD

L'approche de Sawyer et al. (Sawyer et al. 2012b) est implémentée dans le cadre de *Requirements Engineering For self-Adaptive Software systems* (REFAS), un framework multi-vues établi par Munoz et al. (Muñoz-fernández et al. 2015), dans lequel peuvent être implémentés des modèles de variabilité selon des paradigmes différents. Chaque modèle spécifie des exigences, tout en respectant la séparation des préoccupations.
Les modèles des LdPD implémentés dans la version actuelle de REFAS sont des modèles de buts, adaptés aux spécificités de ce domaine. De ce fait, on distingue un modèle de variabilité (fonctionnelle et non fonctionnelle), un modèle de contexte, un modèle de satisfaction de but, et un modèle d'assets. Les concepts intervenants au niveau de chaque modèle sont décrits ci-dessous.

- **Un But (Goal)** fait partie du modèle de variabilité. Il représente les exigences fonctionnelles réutilisables par le système de systèmes. Chaque but est affiné en sous buts, jusqu'à ce que ces derniers deviennent opérationnalisables. Chaque but peut être réalisé de diverses manières dans le système, ce qui peut être formalisé grâce au concept d'opérationnalisation.
- **Une opérationnalisation**, décrite également dans un modèle de variabilité, représente toutes les caractéristiques réutilisables des SdS_Dy. L'activation d'une opérationnalisation implique la satisfaction du but auquel elle est rattachée.
- **Un Soft-Goal** fait référence aux exigences non-fonctionnelles, qui sont partiellement satisfaites, selon l'état et le contexte du SdS_Dy. Chaque soft-goal est satisfait, à un certain degré, par une des opérationnalisations auxquelles il est rattaché.

- **Une Soft-Dependency** représente le niveau de satisfaction requis d'un soft-goal étant donné l'accomplissement d'une condition.
- **Une claim** indique des hypothèses sur la satisfaction de l'opérationnalisation. Elle est considérée vraie jusqu'à ce que le contexte d'exécution indique que l'hypothèse n'est plus maintenue.
- **Une variable de contexte** est une abstraction sur une partie de l'environnement d'un système, et est surveillée au moment de l'exécution par détection.

3.2.3.2. Modélisation du cas GreenLife : Modèles selon l'approche des LdPD

Les exigences fonctionnelles sont les premières à être spécifiées, par le biais du concept de buts. Elles sont ensuite affinées en sous buts si nécessaire. Chaque but est réalisable par un ensemble d'opérationnalisations qui correspondent aux éléments réutilisables de la ligne de produits dynamique. Ils sont reliés à un but par le biais de connecteurs qui décrivent la nature de leur contribution dans les LdPD, c'est à dire qu'ils peuvent être obligatoires, optionnels, régis par une cardinalité de groupe telle que l'exclusion mutuelle, etc. La Figure 3-7 illustre le détail du but « Contrôler », qui peut être réalisé par les opérationnalisations qui lui sont connectées.

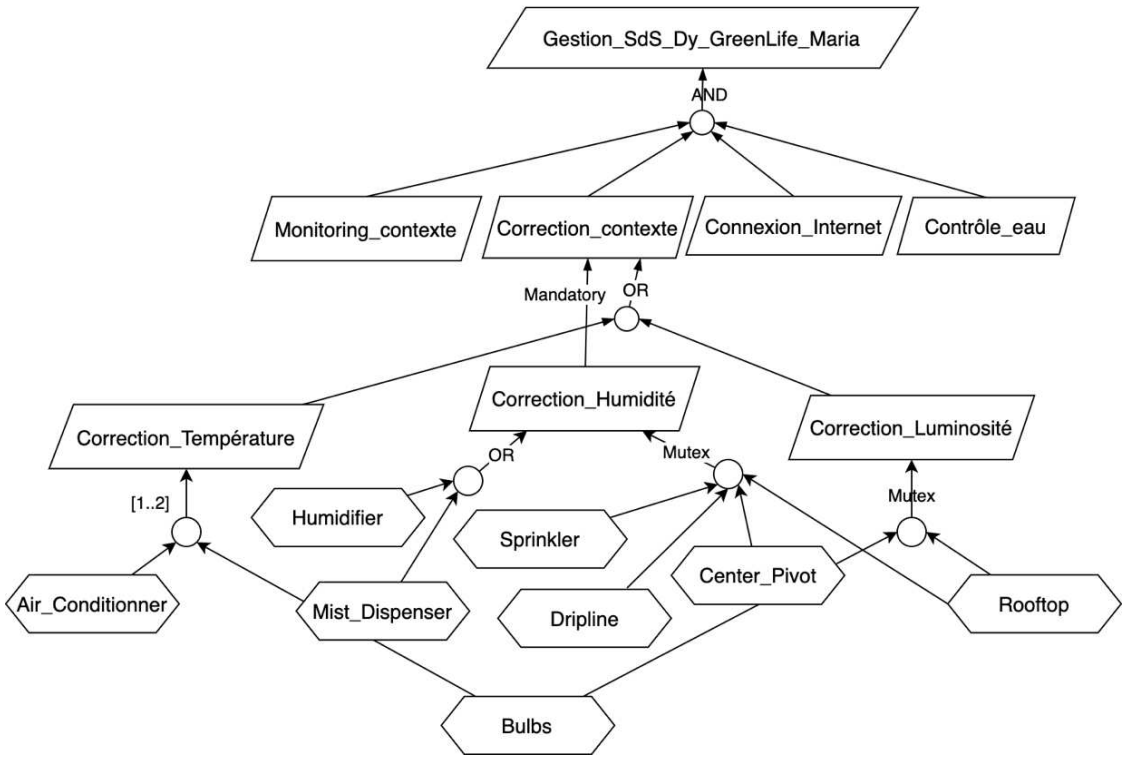

Figure 3-7 : Modèle de variabilité REFAS

L'une des spécificités notable de la notation REFAS est qu'elle permet de tenir compte de l'incertitude de l'environnement. Pour ce faire, chaque opérationnalisation est liée à une exigence non fonctionnelle, de manière à décrire le degré de contribution qu'on attend de celle-ci, et donc de manière variable en fonction des changements de l'environnement. Les exigences non-fonctionnelles sont représentées, comme illustré dans la Figure 3-8 par des soft-goal. La nature de la contribution des opérationnalisations dans la satisfaction (Appelé *satisficing* en anglais) peut être très positive (4), positive (3), neutre (2), négative (1) ou très négative (0).

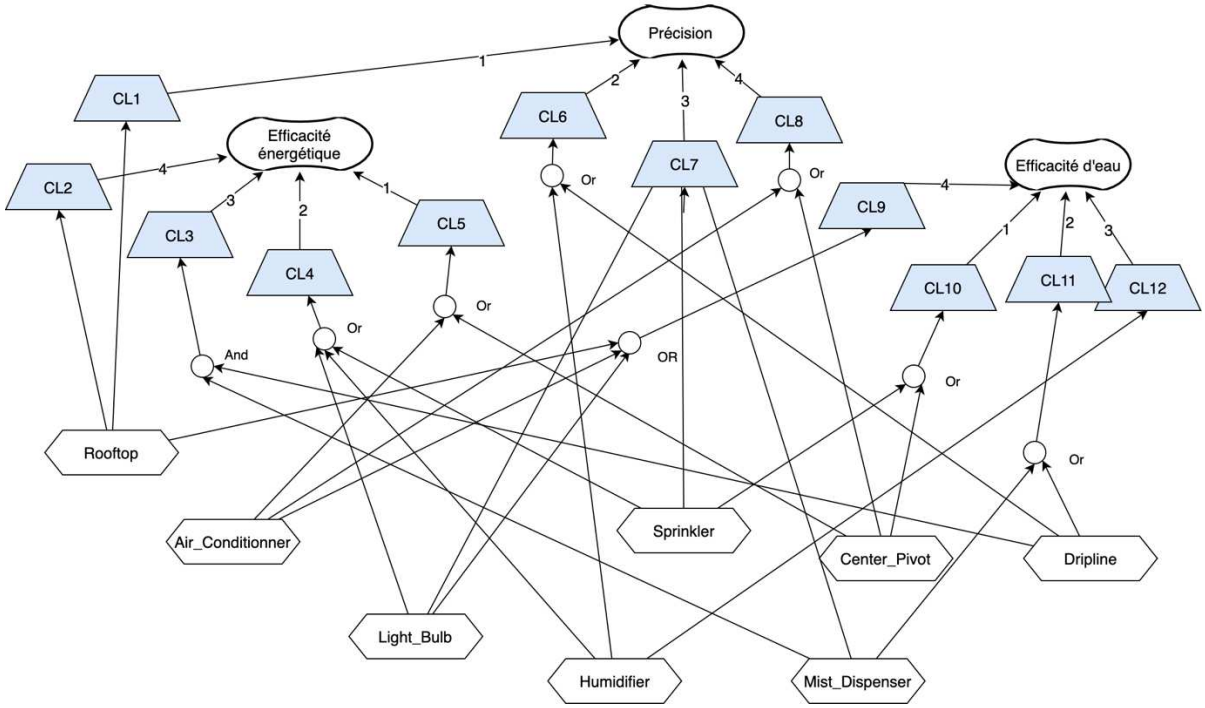

Figure 3-8: Modèle Soft-goal

Afin de spécifier le comportement dynamique de la ligne de produit face aux changements de son environnement, les variables de contexte sont définies dans un modèle de contexte, tel qu'illustré dans la Figure 3-9 (a). Pour chaque instance de celui-ci, les soft-dependencies qui correspondent à l'état actuel du contexte sont maximisées, et les dépendances qui en suivent le sont respectivement. Ainsi, pour chaque changement de contexte, un niveau de satisfaction des exigences non-fonctionnelles est requis, et un nouveau choix d'opérationnalisations est effectué en conséquence. La Figure 3-9 (b) illustre l'impact du contexte sur la satisfaction des soft-goals.

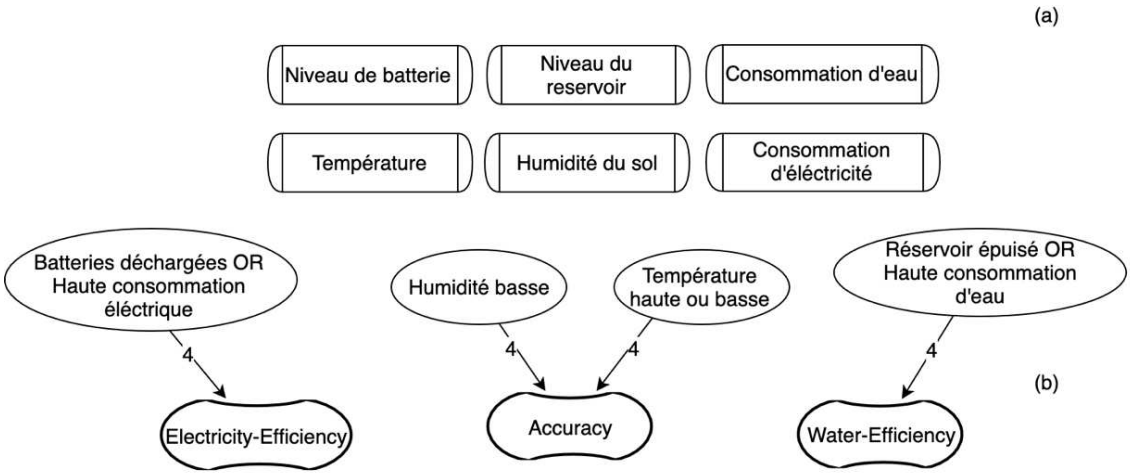

Figure 3-9 : Modèle de contexte et de Soft-Dependencies

Les lignes de produits logiciels dynamiques selon la notation REFAS empruntent le concept de but pour de spécifier les objectifs globaux du système de systèmes dynamique, ainsi que leur affinement en objectifs opérationnalisables. Il guide le choix de la sélection d'une opérationnalisation par le biais de claims, qui présentent une description légère du contexte. Le peu de considération pour la description de l'environnement nécessaire pour déclencher ces changements de configuration demeure cependant problématique, comme il est le cas d'exigences telles que « E_A/H_11 : Quand la consommation d'électricité est normale, et que la luminosité n'est pas haute, ou qu'il pleut et le niveau d'humidité du sol est en dessous de la normal (ElectricConsommation > optimale & Brighntess < High & Rain=False) OR (Rain=True /\ HL>Normal) le SdS_Dy doit activer le rooftop en position fermée ». Certaines caractéristiques de la structure du SdS_Dy, telles que les multi-instances ainsi que les traitements qui en découlent, ne sont pas réalisables non plus.

3.3. Analyse des modèles de spécification de systèmes de systèmes dynamiques

L'étude comparative porte sur trois volets. D'abord, la capacité des modèles à spécifier les exigences de domaine des SdS_Dy à partir desquels des systèmes individuels sont générés est explorée (**QR_L1**). Ensuite, le pouvoir d'expression des notations, par rapport à la typologie d'exigences générée du chapitre précédent, est inspecté (**QR_L2**). Finalement, afin prendre en compte les exigences évolutives provenant de l'utilisateur ou

conséquence d'un changement dans la composition du SdS_Dy, le pouvoir d'évolution de la notation est évalué (**QR_L3**).

3.3.1. Spécification des exigences de différents niveaux d'abstraction

Les exigences de la solution GreenLife sont définies comme une base commune à tous les systèmes dérivés. Elles doivent ainsi être maintenues dans la spécification des systèmes individuels dérivés. A titre illustratif, les dépendances existantes entre le distributeur d'eau « Rainwater » et l'élément « Pump », ou entre les « bulb » et le « Rooftop » ne sont pas explicitement exprimées par Maria. Elles font cependant partie des spécifications de base de la solution GreenLife et doivent ainsi être considérées dans le fonctionnement du système dérivé pour Maria.

- Les modèles de buts sont considérés comme un standard de facto dans la littérature scientifique en ingénierie d'exigences. Ce constat est confirmé par plusieurs auteurs, tel que Ahmad et al. (Ahmad et al. 2012) ou Elsayed et al. (Elsayed et al. 2017). Les modèles de buts assurent en effet une traçabilité entre les objectifs globaux attendus par le SdS_Dy et les exigences qui contribuent à leur réalisation. Ils garantissent également la satisfaction complète ou partielle de ces exigences et assurent la traçabilité avec les différentes formes d'opérationnalisation nécessaires à cet effet. La notation par but permet donc en effet de spécifier les exigences de hauts niveaux, c'est à dire les exigences relatives à la solution générique GreenLife, tout en permettant l'affinement de ces exigences pour spécifier celles qui sont de plus bas niveau, propres à Maria.
- L'ingénierie système (IS) (System Engineering : SE en anglais) est au cœur du développement de systèmes à grande échelle. Les résultats de l'IS et de l'ingénierie de domaine (Domain engineering: DE en anglais) servent comme point d'entrée aux activités d'ingénierie de systèmes logiciels uniques (Favaro and Mazzini 2009). Les relations entre les exigences de haut niveau, ainsi, que les autres éléments du méta modèle SysML qui les vérifient, permettent de maintenir la traçabilité nécessaire pour maintenir les exigences implicites de haut niveau (e.g. la solution générique GreenLife), dans des niveaux d'abstraction plus opérationnels et spécifiques (e.g. Maria).
- La spécification de familles de systèmes est la spécialité de l'ingénierie des lignes de produits logiciels. En effet, ce paradigme permet la conception et développement

d'une variété de produits, à partir d'une base commune d'éléments réutilisables, tout en garantissant qualité et efficacité (Metzger and Pohl 2014). Le processus d'ingénierie des lignes de produits logiciels est distingué d'un côté par l'ingénierie de domaine, dans laquelle sont spécifiées les exigences réutilisables, et d'un autre, par l'ingénierie d'application, dans laquelle sont spécifiées les exigences des utilisateurs finaux. Les exigences de la solution GreenLife sont ainsi des exigences de domaine et sont représentées dans un modèle de variabilité. Les exigences de Maria sont des exigences d'application, et sont utilisées en entrée du processus de configuration pour dériver le produit final (i.e. le SdS_Dy Maria).

En conclusion, les trois approches permettent la spécification des exigences de la solution GreenLife et celles de l'utilisatrice Maria. La première approche à travers l'affinement des buts en exigences, dans le modèle de buts. La deuxième approche à travers le modèle d'exigences qui décrit les exigences d'un système de systèmes, dans ses différents niveaux d'abstraction (i.e. le SdS_Dy, les sous-systèmes, les composants logiciels, etc). Et finalement, le troisième à travers deux processus d'ingénierie, de domaine et d'application, afin de spécifier respectivement des exigences réutilisables, et des exigences de clients finaux.

3.3.2.Spécification de la typologie d'exigences variées

Les exigences tirées du cas GreenLife représentent un échantillon non exhaustif de la typologie d'exigences susceptibles d'être décrites puis spécifiées formellement pour concevoir des SdS_Dy. Les notations des modèles de buts, SysML et des LdPD ont été analysées, afin de révéler leurs principaux atouts et détecter les limitations qui doivent être rectifiées.

3.3.2.1. Spécification de la typologie d'exigences des SdS dynamiques selon l'approche par buts

L'usage d'un modèle de buts pour la spécification de systèmes de systèmes dynamiques permet en effets la définition d'exigences fonctionnelles et non fonctionnelles d'application, et ce respectivement via les concepts de Goal et Soft-Goal. Certaines dérivations de la notation par but utilisent des dépendances numériques entre buts, pour définir les degrés de satisfaction nécessaires ou exigés. La hiérarchisation de ces

exigences dans un SdS_Dy, et la représentation de dépendances transverses apparaissent également dans les modèles de buts. Cependant, leur spécification demeure contraignante, principalement pour décrire les relations entre exigences et opérations (e.g. Le SdS dynamique doit garantir l'irrigation du champs par au moins un parmi les moyens d'irrigation possibles (Sprinkler, Dripline, Rooftop)) et entre les opérations elles-mêmes (e.g. Le SdS_Dy doit activer les Dripline avec une pression respective de 15,30 et 40 psi pour des nombres d'émetteurs égale à 114, 200 et 232). Seules des relations de type AND et OR existent et limitent la représentation de la structure hiérarchique au raffinement de buts. Cette limitation est aussi observée au niveau des relations hiérarchiques des opérationnalisations, qui nécessitent bien plus que le « OR » et « AND » pour définir leur rapport aux exigences et leurs dépendances mutuelles. En effet, chaque exigence est connectée à un ensemble d'opérationnalisations qui décrivent le comportement potentiel du système concerné par l'exigence, lorsque celle-ci doit être satisfaite. Selon la typologie des exigences des SdS dynamiques, ces connections doivent décrire des relations obligeante, optionnelle, de groupe et aussi de nombre d'instances de ces opérationnalisations, chose impossible par une simple relation « OR » ou « AND » Par ailleurs, le type d'une opérationnalisation qui est représentative d'une partie de la configuration du SdS, n'est pas que booléen (opérationnalise ou pas une exigence), il peut également être numérique ou même paramétrique (instancié en build-time). Cette typologie n'est cependant pas prise en compte dans la notation par but étudiée. Cette différenciation est importante dans la spécification des exigences d'application, principalement les exigences de préférences qui spécifient non seulement les fonctionnalités requises ou dépréciées du SdS dynamique, mais aussi des valeurs ou optimums exigés pour certaines de ces fonctionnalités. Les relations traverses se limitent eux aussi à des relations d'exclusion mutuelle décrivant un conflit. Néanmoins, les dépendances entres les opérationnalisations des SdS dynamiques peuvent également être arithmétiques ou logiques. Ces dernières n'étant pas prises en compte dans la notation par buts présente donc une restriction. L'impact du contexte sur le SdS_Dy est capturé par le concept « Événement » qui décrit des événements ponctuels qui impliquent l'usage d'une opérationnalisation au lieu d'une autre. Le pouvoir d'expression pour la spécification du contexte de ce type de systèmes est très élémentaire et insuffisante. Des expressions composées telles que ((ConsumptionE>optimale/\Brightness>=High) \/ (ConsumptionE < optimale /\ Brightness >= Low)) /\ (rain=false\/HL<Normal)) ne

sont pas possibles à spécifier par exemple. Les évènements temporels sont pris en compte lorsqu'il s'agit de spécifier un moment précis dans le temps. Cependant, la spécification de processus répétitifs, ou bien de l'ordre avec lequel certains processus doivent être déclenchés n'est pas possible non plus. L'optimisation est représentée dans des modèles de buts par le biais de soft goals, ou de degrés de satisfactions qui relient les opérationnalisations aux buts. Plus ce degré est élevé, plus le but est susceptible d'être complètement achevé, et vice versa. Les exigences d'adaptation font l'objets d'extensions pointues dans des modèles de goal, notamment, les extensions proposées par (Souza et al. 2013) et (Fredericks et al. 2014), pour spécifier des exigences de AwrRqs et EvRqs, ou encore les exigences Relaxables. Cependant, ces extensions sortent du périmètre de cette étude. Par conséquent, les exigences d'adaptation mentionnées ne sont pas spécifiées par les modèles de buts classiques. Les exigences paramétriques, de coût, de préférence et de proportionnalité ne sont pas prises en compte.

3.3.2.2. Spécification de la typologie d'exigences des SdS dynamiques selon l'approche SysML

Les modèles SysML décrivent, par le biais de modèles d'exigences et de cas d'utilisation, toutes les exigences attendues du SdS_Dy. Cependant, bien que les exigences non-fonctionnelles jouent un rôle significatif dans la mise en place de tout système informatique, particulièrement dans la spécification de SdS_Dy qui nécessite souvent de trouver des compromis entre précision et efficacité, leur représentation dans les modèles d'exigences, bien que valable, n'est pas effective et se limite à son énoncé et à la définition de contraintes de performances au niveau des modèles de définition de blocs. Des extensions des modèles SysML par des concepts de modèles de buts permettent en effet la spécification de ce type d'exigences (Tueno et al. 2017), mais sortent du périmètre de cette étude. Le lien entre les exigences et leur opérationnalisation physique et logique (représentés respectivement par le biais de modèles de structure ou de comportement), est spécifié au moyen d'un modèle d'allocations. Toutes les exigences structurelles sont représentées dans des modèles de définition de blocs. En effet, la hiérarchie du SdS_Dy peut être représentée sous forme de blocs ; les propriétés numériques de chaque système sont définies dans les compartiments « Parts » ou « Values », les blocs obligatoires, relatifs aux éléments obligatoires, sont représentés par une composition, les blocs optionnels sont définis grâce aux agrégations, les multi-instances par les associations avec

cardinalités et les relations de groupe par des associations de groupe. Les dépendances «requires» et «excludes» sont respectivement représentées par les annotations portant la même terminologie.

L'environnement, ainsi que l'impact qu'il a sur la composition et le comportement du SdS_Dy, sont modélisés dans des modèles de comportement, principalement, dans des modèles de machines à états, d'activité, ou de séquences. Les valeurs de paramètres impliqués dans la modélisation du contexte peuvent être spécifiées dans des compartiment au niveau des blocs, comme ils peuvent être représentés séparément, dans des modèles paramétriques. Spécifiquement, les comportements déclenchés suite à un évènement temporelle sont spécifiés dans des modèles de machine à états ou d'activités. L'ordonnancement des processus, lui, est représenté dans un modèle de séquences. Les exigences de coût, quant à elles, peuvent être spécifiées dans un modèle paramétrique. Tout comme le modèle de but, les exigences d'adaptation (e.g : Les niveaux de batterie des capteurs d'humidité de doivent pas être en dessous de 40%, le cas échéant, le nombre de capteurs actifs doit être réduit) ne sont pas centraux aux modèles SysSML, cependant, certaines extensions, bien que non considérées dans cette étude, permettent leur spécification, tels que (S. M. Lee et al. 2019) pour la représentation des exigences AwRqs. Le langage SysML ne contraint pas le système pour spécifier des exigences de proportionnalité (e.g : Le SdS_Dy doit utiliser un nombre d'ampoules actif proportionnellement à la température capturée), de préférence (e.g : Le SdS_Dy doit utiliser le toit transparent), d'autonomie (e.g : Le SdS_Dy doit muter entre le toit et les ampoules pour assurer le control d'humidité) ou d'optimisation (e.g : Lorsque la consommation d'eau est élevée, le SdS_Dy doit minimisez les Sprinkler actifs).

3.3.2.3. Spécification de la typologie d'exigences des SdS dynamiques selon l'approche des LdPD

Les exigences fonctionnelles et non fonctionnelles sont spécifiées de la même manière que dans les modèles de buts, à travers les concepts de *Goal* et *Soft Goal.* De même pour les exigences d'optimisation qui sont spécifiées dans les modèles de *Soft-Dependencies*, décrivant le degré de satisfaction des qualités attendues du SdS_Dy dépendamment du choix de l'opérationnalisation. Le modèle de variabilité est principalement efficace dans la spécification des exigences structurelles du domaine. En effet, les caractéristiques booléennes, qui peuvent ou pas être activées dans une configuration donnée, et qui sont

contraintes par des dépendances obligatoires, optionnelles ou de groupe, peuvent être représentées dans un modèle de variabilité. Ce type de relations est possible au niveau de l'affinement des buts en sous buts, mais aussi au niveau des opérationnalisations de ces buts. Les relations horizontales sont également représentées dans ce même modèle, afin de définir les exigences et les opérationnalisations qui s'excluent, ou qui se sélectionnent en pairs. Les éléments numériques, notamment ceux définis dans un domaine restreint, ne sont néanmoins pas représentés (e.g. Le SdS_Dy peut activer les Sprinkler avec une rotation de 90°, 180°, 240° or 360°). Les opérationnalisations qui existent en plusieurs occurrences, ne sont pas représentées non plus dans cette version du modèle des LdPD. La gestion du comportement dynamique est possible grâce aux *soft-dependencies* d'un côté, et aux *claims* d'un autre. Les deux sont alimentés par les valeurs des variables de contexte. Dès lors, les exigences contextuelles sont spécifiées avec succès, et les exigences de composition le sont implicitement. Les événements qui relèvent du temps sont plus compliqués dans leur spécification, en particulier, les exigences qui doivent être satisfaites selon une fréquence donnée (e.g. Le SdS_Dy doivent connecter les dispositifs au réseau Wi-Fi chaque 48h), ou dans un ordre particulier (e.g. Le SdS_Dy doivent fermer le toit avant d'activer les ampoules). En outre, il n'existe pas de mécanisme qui représente une contrainte directe sur les règles de sélection d'une opérationnalisation donnée, pour un contexte spécifique. Les exigences paramétriques, de proportionnalité, de coût ou bien de préférence ne sont en revanche pas représentées.

Au bout du compte, aucune des notations ne permet la spécification exhaustive de toutes les exigences de la typologie déduite du cas GreenLife. Cependant, l'analyse des trois notations met en évidence les points de forces et les faiblesses de chacune. Nous distinguons particulièrement le pouvoir d'expression des modèles des LdPD dans la représentation de la variabilité, et celui de la notation SysML dans la représentation du comportement dynamique, par l'intermédiaire du modèle de machines à états finis.

3.3.3. Spécification des exigences dynamiques

Les exigences décrites pour une application donnée sont représentatives de ce qui est attendu d'un système à un moment donné. Cependant, ces exigences sont susceptibles de changer, non seulement en raison du changement des besoins de l'utilisateur lui-même, ou de la composition du SdS_Dy, mais aussi en raison des adaptations nécessaires lors d'évolutions de l'environnement. Les conséquences de telles évolutions ont des

implications sur la structure et le comportement du SdS_Dy et sur les dépendances entre ces exigences. Par exemple, suite à un premier retour d'expérience, Maria décide d'utiliser son SdS_Dy différemment, pendant les saisons d'automne et d'hiver, et entre les saisons du printemps et de l'été. Parmi les conséquences de ce choix, il y a lieu de distinguer les capteurs d'humidité qui ne peuvent être plus qu'au nombre de 3 en mode master dans le premier cas de figure, mais peuvent aller jusqu'à 7 instances, en mode Master, dans le deuxième. Cet exemple décrit deux exigences qui, bien que simples, ne peuvent être spécifiées par aucune des notations présentées plus haut.

3.4. Conclusion

L'étude comparative réalisée dans ce chapitre vise à évaluer le pouvoir d'expression des notations par but, SysML et des LdPLD, et ce, pour la spécification de systèmes de systèmes dynamiques. Le cas de la solution GreenLife déployé pour la cliente Maria est utilisé pour illustrer les propos soutenus.

Les principaux résultats de cette étude comparative sont décrits dans le Tableau 3-1. Les (+), (≈) et (-) symbolisent respectivement les exigences qui sont complétement, partiellement ou non spécifiées par l'approche en question.

Typologie d'exigences			**Selon l'approche par buts**	**Selon l'approche SysML**	**Selon l'approche des LdPD**
Exigences de domaine	Segment de marché	Fonctionnelle	+	+	+
		Non Fonctionnelle	+	-	+
	Variabilité	Structure	+	+	+
		Multiplicité	-	≈	≈
		Dépendances	-	+	+
Exigences d' application	Fonctionnelle		+	+	+
	Non Fonctionnelle		+	+	+
	Optimisation		+	-	+
	Préférence	Booléen	+	+	+
		Paramétrique	-	+	-
	Autonomie		+	+	+
	Cout		≈	+	-
	Proportionnalité		-	-	-

Exigences d' adaptation	Relaxable	-	-	+
	Optimisation	+	-	+
	Temporelle	≈	+	≈
	AwRqs	-	-	-
	EvRqs	-	-	-
	Contextuelles	+	+	+

Tableau 3-1: Récapitulatif des approches de spécification de SdS_Dy

Les modèles de buts sont efficaces pour la spécification de systèmes réactifs, du fait qu'ils permettent la modélisation des systèmes en question, mais aussi de leur environnement et de l'impact qu'il a sur ce premier. Cependant, ces modèles ne sont pas appropriés pour spécifier les dépendances entre exigences, notamment à des fins de réutilisation. Celles-ci étant importantes surtout pour la représentation des exigences de domaine. La spécification formelle des exigences de comportement, au niveau de l'application ou de l'adaptation de celle-ci, telles que les exigences paramétriques, temporelles ou encore de proportionnalité n'est pas possible non plus. Les modèles des LdPD sont principalement efficaces pour la spécification des exigences de domaine. Ils prennent en compte toutes les exigences potentiellement mises en vigueur par un SdS_Dy, ainsi que les différentes relations entre elles. Cependant, bien que ce paradigme permette bien la spécification d'exigences d'application et d'adaptation, il est limité dans le sens où les états possibles du SdS_Dy, ainsi que les événements qui les activent, ne sont pas explicitement et efficacement spécifiées (Reinhartz-Berger et al. 2011). Ce concept est en effet supporté par le langage SysML grâce aux modèles machines à états finis, qui ont un grand pouvoir d'expression. Ces modèles spécifient l'évolution du comportement d'un système selon l'évolution de son contexte. SysML reste cependant focalisé sur les aspects opérationnels que de configuration. De ce fait, bien qu'il admette la spécification des exigences de domaine et d'application, par le biais de modèles de structure tels que les modèles de blocs, la nature reconfigurable des sous-systèmes, conséquence de la versatilité de ce domaine, n'y est pas une préoccupation de premier ordre. Pour conclure, un langage pour la spécification des systèmes de systèmes dynamiques adéquat devrait prendre en compte les atouts apportés par l'ingénierie des lignes de produits logiciels dynamiques, tout en considérant l'évolution potentielle des exigences. Le recours aux concepts des machines à états finis, s'avère un choix judicieux pour représenter cette évolution.

Partie II - Spécification des exigences de variabilité

Chapitre 4 Cadre de travail

4.1. Introduction

La spécification des SdS_Dy est complexe dans la mesure où en plus des exigences inhérentes à ce domaine, des exigences de variabilité se manifestent à plusieurs niveaux. Le SdS_Dy doit ainsi être capable de détecter cette variabilité quand elle se présente et de planifier par conséquent les adaptations nécessaires.

La création de ces systèmes dynamiques n'est pas une préoccupation complètement nouvelle dans la recherche. En fait, plusieurs approches et Framework ont été développés au fil des années pour offrir ces propriétés d'adaptabilité. On distingue notamment les approches basées sur les architectures, qui formulent et traitent les changements au niveau de modèles architecturaux (Denko et al. 2009), les approches basées sur des agents, qui modélisent les systèmes comme un ensemble d'agents autonomes (Filipe et al. 2011) ou les approches réflexives, qui offrent les outils nécessaires pour observer et modifier la composition de systèmes au moment de l'exécution (Baumer et al. 2014). Cependant, il a été soutenu (Salinesi et al. 2010; Raúl et al. 2012; Dumitrescu et al. 2013a; Raúl et al. 2014; Achtaich et al. 2018; Triki et al. 2015) que la gestion efficace de la variabilité, qu'elle soit statique ou dynamique, peut se faire via une démarche exploitant la programmation par contraintes comme fondement théorique et concept central pour la résolution de l'ensemble des problèmes rencontrés. Il s'agit de s'inscrire dans une approche dirigée par des modèles (Rouvoy et al. 2009; Raúl et al. 2015; Villota et al. 2018), qui utilise des modèles de variabilité est traduit en programmes de contraintes pendant l'exécution, pour dériver des produits valables dans des contextes différents. Ceci étant, une implémentation du Framework des LdPD, adaptée aux besoins et défis spécifiques aux systèmes de systèmes dynamiques, s'avère nécessaire.

Ce chapitre décrit les différents types de variabilité perçus dans les SdS_Dy, en illustrant avec le cas du système d'irrigation. Le chapitre introduit les dimensions d'un SdS_Dy, avant de présenter le cadre des LdPDs adapté, ultérieurement élaboré dans cette thèse.

4.2. Variabilité multidimensionnelle

L'étude de cas détaillée dans le Chapitre 2 démontre la diversité des exigences susceptibles d'être décrites, lors de la spécification d'un SdS_Dy. Au-delà des exigences de produit classiques (exigences fonctionnelles , exigences non fonctionnelles, exigences d'interface), des exigences spécifiques à la variabilité apparaissent, comme le montrent

(Djebbi and Salinesi 2006) (Muñoz-Fernández et al. 2018). La variabilité fait référence à la capacité d'un système à être paramétré, reconfiguré, personnalisé ou étendu pour être utilisé dans des contextes spécifiques (Bachmann and Clements 2005). Les exigences pouvant concerner aussi bien le besoin de diversité que le besoin de spécificité face à la diversité, la gestion des exigences de variabilité consiste donc à traiter les divergences et similitudes entre systèmes ou entre dispositifs, ainsi que le paramétrage, la réutilisation des composants, la prise en compte des contextes hétérogènes, la définition de versions, la spécification de spécificités pour des utilisateurs distincts, etc. Dans le domaine des SdS_Dy, cette variabilité se manifeste à plusieurs niveaux et selon différentes facettes (notamment : statique ET dynamique, dans le temps ET dans l'espace, ainsi que interne ET externe). Une modélisation cohérente et adaptées des différentes niveaux et facettes du SdS_Dy semble nécessaire (Raúl Mazo 2014) (Bürdek et al. 2013), chose proposée dans les sections suivantes.

4.2.1. Variabilité statique

La variabilité est la possibilité de modifier ou de personnaliser un système (Van Gurp et al. 2001). Dans le domaine des SdS, cette variabilité est fréquente, principalement, à cause de la nature imbriquée et hautement personnalisable de ces systèmes. Cependant, grâce à cette variabilité, il est possible de concevoir une panoplie d'applications qui répondent aux besoins divergents des utilisateurs finaux. Le type d'affichage, le protocole de communication, le processus d'identification ou encore la nature et nombre de dispositifs imbriqués sont des exemples de points de variation souvent constatés dans un SdS_Dy. Ces possibilités sont prévues au préalable, lors de la conception initiale du SdS_Dy.

Exemple 4.1 :Outre les composants de base, un capteur d'humidité intelligent peut être fabriqué à partir de différents composants, notamment, des capteurs de pollution, des clignotants, des alarmes, des antennes ZigBee, etc.

4.2.2. Variabilité dynamique

L'expérience montre qu'une configuration statique d'un SdS_Dy est insuffisante et peut aboutir à des résultats qui ne sont pas correctement alignés avec les besoins, voire carrément indésirables (L'Obs 2016). Ceci est principalement dû au contexte changeant et parfois imprévisible dans lequel ces systèmes sont déployés et opèrent, pouvant ainsi

affecter leur utilité ou leur performance. Ainsi, la variabilité dynamique fait référence aux différentes configurations que peut prendre un système de systèmes, face aux divers changements dans le contexte après l'exécution, ces changements n'ayant pas été identifiés à priori (Ortiz et al. 2012; Cetina et al. 2009).

Exemple 4.2: Lorsque la consommation d'eau s'approche de la consommation maximale, l'irrigation se fait par le biais des sprinklers 1,2, 3 et 4 utilisent la tête Rotor avec une rotation 240° au lieu d'activer tous les sprinkler (9 instances), avec une tête Spray et une rotation 45°.

4.2.3. Variabilité dans l'espace

La variabilité dans l'espace est définie par Pohl et al. comme étant l'existence de différentes versions d'un même artefact, toutes valides à un moment donné (Pohl et al. 2005), par exemple pour différents clients (correspondant à différentes organisations), ou pour différents utilisateurs (au sein de la même organisation). Cette propriété des familles de produits se retrouve dans SdS_Dy. Par définition, les systèmes qui constituent les SdS_Dy sont susceptibles d'exister sous plusieurs formes et de fonctionner avec des modes distincts dans des SdS différents. La variabilité dans l'espace offre ainsi la possibilité d'adapter la configuration de ces systèmes aux besoins de différents utilisateurs.

Exemple 4.3: L'irrigation peut être automatisée grâce aux dispositifs «Dripline», «Sprinkler» et «Rooftop». Le SdS_Dy peut modifier le moyen d'irrigation à employer dépendamment de son contexte.

4.2.4. Variabilité dans le temps

La variabilité dans le temps apparait lorsque différentes versions d'un artefact sont valides chez le même client, pour le même utilisateur, à des moments distincts (Pohl et al. 2005) (C. Elsner et al. 2010). Elle est liée à l'évolution d'un SdS_Dy, qu'il s'agisse d'évolution technologique, structurelle, fonctionnelle, ou en termes d'exigences. La variabilité dans le temps peut concerner tous les aspects du SdS et des système qui les composent, aussi bien les protocoles de communication, que les mécanismes d'identification, les systèmes d'exploitation, des dispositifs nouveaux, etc. Sur le plan technologique, la variabilité dynamique fait généralement référence à des points de

variation au niveau desquels des variantes sont remplacées par de nouvelles variantes, choisies par exemple parce qu'elles sont jugées plus performantes. Afin d'accompagner cette évolution technologique et de satisfaire les besoins progressifs des utilisateurs qui sont eux aussi dynamiques, les exigences pour la spécification de SdS_Dy varient aussi dans le temps.

Exemple 4.4: Suite à des efforts de standardisation de communication pour l'IdO, un nouveau protocole «ComPourTous» a été mis au point. Par conséquent, les variantes «Wi-Fi», «Bluetooth», et «ZigBee», appartenant au point de variation «Technologie de communication», sont remplacées par le composant «ComPourTous».

4.2.5.La variabilité interne

La variabilité interne (Pohl et al. 2005) concerne des artefacts qui ne sont pas forcément liés à des choix directs des utilisateurs, mais qui offrent aux SdS_Dy, la flexibilité nécessaire pour déterminer, parmi un ensemble de possibilités, l'option la plus performante, optimale ou adéquate aux exigences ou à un contexte donné.

Exemple 4.5: Le point de variation «Rain_Broadcast », dérivée en variantes « time_space » et « instant », ne correspond pas à un choix direct de l'utilisateur, mais plutôt à un moyen pour garantir la précision des données collectées.

4.2.6.La variabilité externe

La variabilité externe (Pohl et al. 2005), est une variabilité perçue par l'utilisateur. Elle permet aux différents utilisateurs de composer et configurer un SdS, différemment, selon leurs préférences personnelles.

Exemple 4.6: Les clignotants et les alarmes sont imbriqués aux capteurs d'humidité, sur la demande de l'utilisateur.

4.3. Gestion de la variabilité

La gestion de la variabilité rend possible le développement de familles d'applications (Jazayeri et al. 2000), en exploitant les similitudes entre ces dernières. Cette approche permet la création de solutions pour des cas d'utilisation spécifiques, tout en minimisant

les coûts de développement. La réutilisation systématique de la connaissance dans un domaine d'application, centrale à l'activité de gestion de la variabilité, est en effet un moyen puissant pour marier performance, optimisation et flexibilité. La création d'une application, à partir de la famille de produits, consiste à adapter ses éléments communs et variables, dépendamment du contexte dans lequel l'application est déployée.

4.3.1. Ingénierie des lignes de produits logiciels

La représentation et la gestion de la variabilité sont au cœur de l'ingénierie des lignes de produits logiciels, qui consiste d'une part à développer et à construire la base commune et variable d'artefacts réutilisables et d'autre part à les utiliser pour la construction de produits spécifiques, à partir de la ligne de produits logiciels. Pour ce faire, deux processus d'ingénierie complémentaires sont adoptés, tel qu'illustrés dans la Figure 4-1.

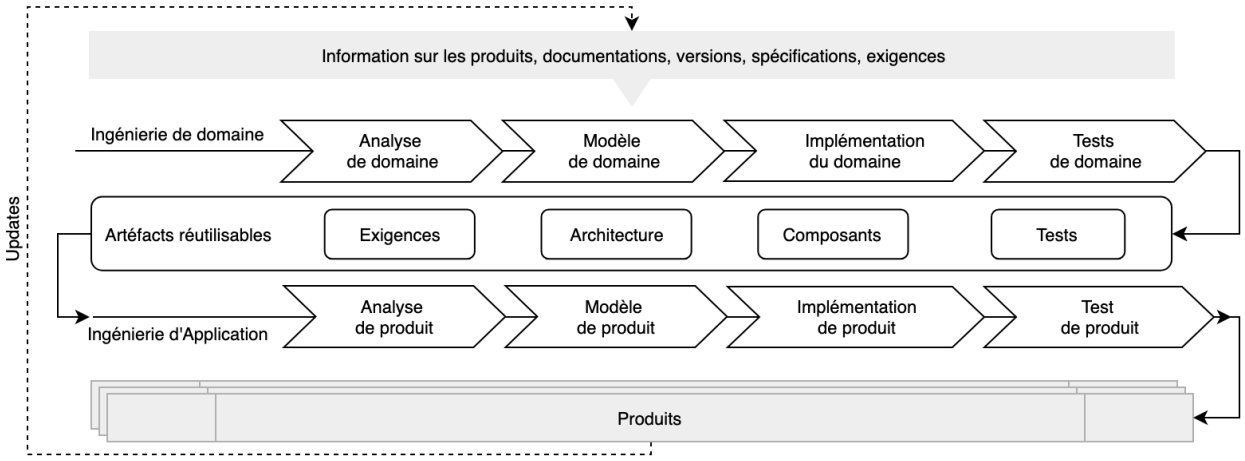

Figure 4-1: Ingénierie des lignes de produits logiciels

- *L'ingénierie de domaine* est une phase de développement *pour* la réutilisation. C'est une approche systématique pour identifier les éléments communs et variables -dans l'espace et dans le temps, des produits, notamment en termes d'exigences, d'architectures, de composants, etc. mais aussi d'environnement ou de caractéristiques des utilisateurs (Czarnecki 1998). Réalisée par des experts, l'analyse du domaine a pour résultat la représentation des connaissances relatives au domaine au sens large, mais aussi particulièrement aux produits à concevoir et développer. Elle offre donc en particulier une description ''générique'' de tous les artefacts ainsi que des dépendances qui les relient et aux règles et contraintes qui régissent leur

réutilisation, fournissant ainsi les artefacts nécessaires pour une approche systématique de réutilisation, de la conception à implémentation et aux tests.

- *L'ingénierie d'application* est une phase de développement *par* la réutilisation. Cette phase exploite les artéfacts produits par l'ingénierie de domaine, en s'assurant que la construction des produits est conforme aux exigences, au contexte d'utilisation visé, mais aussi aux dépendances, règles et contraintes spécifiées au niveau du domaine. Il s'agit donc de sélectionner puis d'assembler puis les artefacts réutilisables en partant des besoins des utilisateurs. Le résultat de cette activité est multiple : produit dérivé, application exécutable, architecture, unité de test, etc. Cette dérivation est dite *statique* lorsqu'il s'agit d'une adaptation manuelle d'artefact. Elle peut être qualifiée de *dynamique*, dans le cas d'une auto-adaptation en cours d'exécution.

4.3.2. Lignes de produits logiciels statiques

Une ligne de produits est un ensemble de produits qui, ensemble, s'adressent à un segment de marché particulier ou remplissent une mission donnée. Les produits dérivés répondent aux besoins spécifiques des clients auxquels ils sont destinés, tout en partageant une base commune (Clements and Northrop 2002). Une ligne de produit est dite statique lorsqu'elle est conçue pour que l'ingénierie d'application soit réalisée ''at build time'', c'est-à-dire une et une seule fois, au préalable de l'exécution, et sans que cela soit encore possible en cours d'exécution.

Dans le cas de la ligne de production de GreenLife, chaque système d'irrigation, dont celui de Maria, rassemble des éléments de manière unique. La sélection de ces éléments, qui résulte de choix d'options comme la nature et les modèles des dispositifs qui la composent, est réalisée au moment de l'achat d'un système par un client et de leur configuration. Si la ligne de produits GreenLife n'était définie qu'au travers de ces types de choix d'option, elle serait statique. Le fait que d'autres types d'options, telles que la capacité de communiquer ou de mesure l'humidité, qui peuvent être activés ou désactivés de manière autonome, en cours d'exécution - qui plus est à l'initiative du système lui-même de manière autonome, montre toutefois que GreenLife n'est en fait pas une ligne de produit statique.

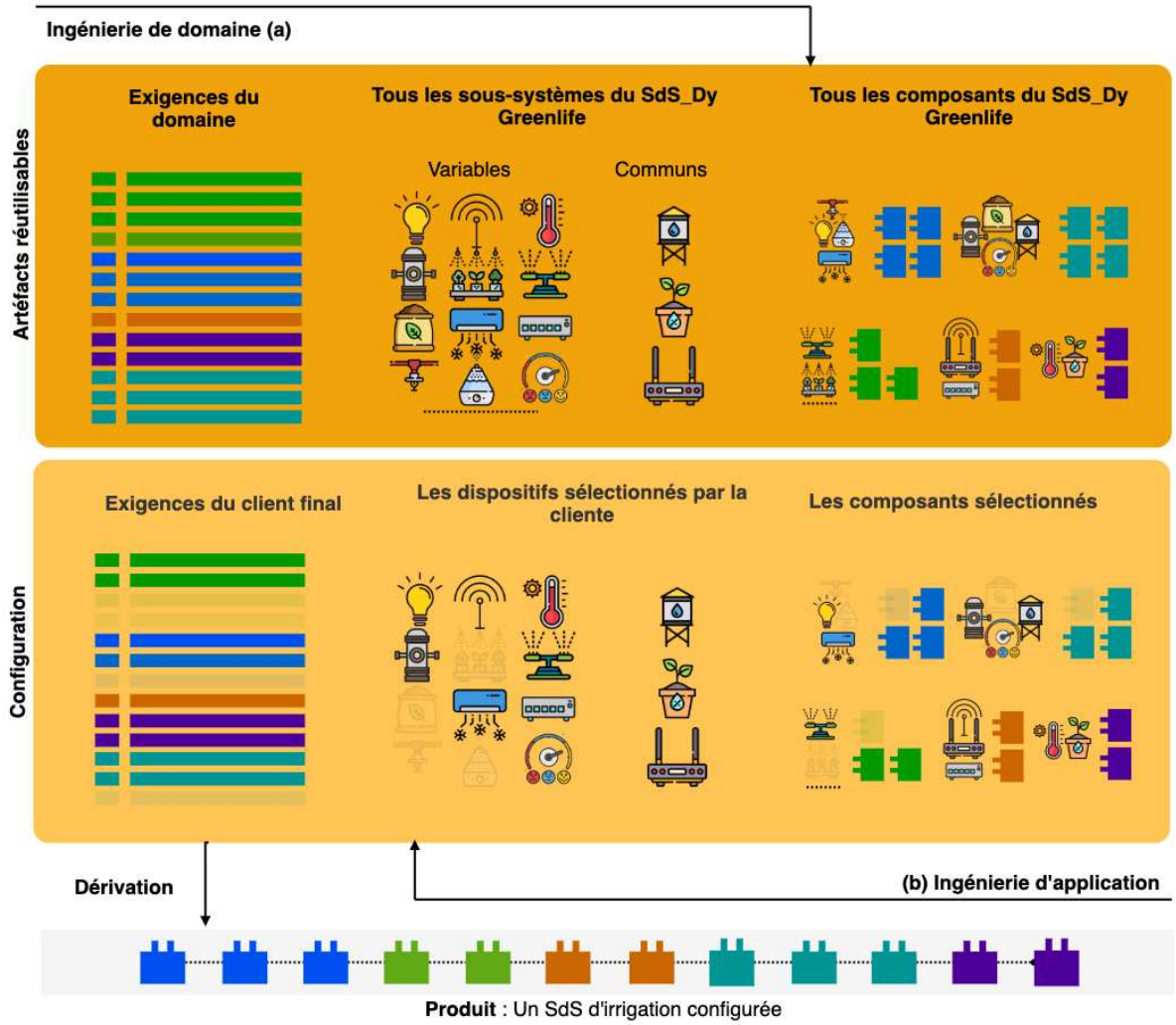

Figure 4-2: Processus de dérivation selon le processus d'ingénierie des LdP

Un produit dérivé d'une LDP partage des éléments communs avec d'autres produits, tout en maintenant ses spécificités et en respectant les besoins de l'utilisateur. Cette approche assure une qualité supérieure, un coût inférieur et un temps de développement plus court et mieux maitrisé pour chaque produit (Pohl et al. 2005).

Dans le cas des SdS_Dy, les lignes de produits logiciels peuvent, sur la base d'un ensemble de systèmes imbriqués et d'un espace de configurations -communes et variables- de chaque système, construire des SdS_Dy adaptées aux besoins spécifiques d'utilisateurs distincts. La Figure 4-2 illustre le processus de dérivation d'un SdS, pour un utilisateur donné.

a) En premier lieu, une étude minutieuse du domaine en question aide à définir les qualités que doit satisfaire le SdS, tout en spécifiant la variabilité et les points de variation. Le résultat d'une étude de domaine est l'ensemble des configurations possibles pour le SdS_Dy.

b) En second lieu, à la lumière des exigences de l'utilisateur, la sélection puis l'ajustement et l'assemblage des systèmes sont effectués, afin d'en dériver une configuration satisfaisante, prête à être testée puis déployée.

L'ingénierie des lignes de produits logiciels est cependant insuffisante pour la spécification de SdS_Dy, car les différents types de variabilité, introduits dans la section 4.2, persistent même après la dérivation de la configuration d'un SdS. Des mécanismes de gestion de la variabilité dynamique sont en conséquence indispensables.

4.3.3. Lignes de produits logiciels dynamiques

Les lignes de produits logiciels dynamiques peuvent être configurées en cours d'exécution (Hallsteinsen et al. 2008). Lorsque cette configuration est réalisée à l'initiative du système lui-même et de manière autonome, on parle de système auto-adaptatif. Il est à noter que tous les systèmes auto-adaptatifs ne sont pas conçues comme des lignes de produits dynamiques, et que par ailleurs, toutes les lignes de produits logiciels dynamiques ne sont pas nécessairement auto-adaptatives (on peut par exemple penser aux progiciels paramétrables manuellement en cours d'exécution).

Les lignes de produits dynamiques auto adaptatives sont capables d'ajuster leur propre structure, leur composition ou même leur configuration à la lumière de changements observés dans leur contexte d'exécution (Capilla et al. 2014). En d'autres termes, à chaque changement significatif du contexte, une nouvelle combinaison d'éléments communs et variables est créée, correspondant ainsi formellement à un nouveau produit constitué de produits avec de nouvelles configurations (Sawyer et al. 2012a) (Alférez et al. 2014).

On peut considérer que la dérivation issue d'une ligne de produit dynamique n'est jamais définitive, puisque contrairement au cas des lignes de produits statiques, il est prévu que toute configuration puisse être modifiée à tout moment. La mécanique de mise en œuvre implémente la boucle MAPE (IBM 2005) : les changements qui proviennent du contexte externe, internes, ou même temporels, sont identifiés et analysés. Dès que les exigences ne sont plus satisfaites (par rapport au contexte), une nouvelle dérivation de la ligne de produits est conçue, selon un processus d' « Ingénierie d'adaptation » (Raúl Mazo 2018). Un nouveau choix de variantes et une nouvelle configuration, sont alors valides.

Le processus d'adaptation des LdPD est illustré pour le cas GreenLife dans la Figure 4-3. Sur la base des éléments réutilisables définis par GreenLife pendant la phase d'ingénierie de domaine, une première dérivation est réalisée par Maria dans le cadre d'une phase

d'ingénierie d'application, avec le choix du sprinkler, et de la toiture ouverte. Suite à un changement du contexte, une nouvelle dérivation a lieu, afin d'utiliser plutôt le compte-gouttes comme dispositif d'irrigation, et le climatiseur comme régulateur de température. Un autre changement de contexte requière ensuite une troisième dérivation dans laquelle le compte-gouttes est conservé pour l'irrigation, mais qui fait plutôt usage des ampoules pour régulariser la température. La conception de la LdP en vue de ces adaptations successives relèvent de l'ingénierie d'adaptation.

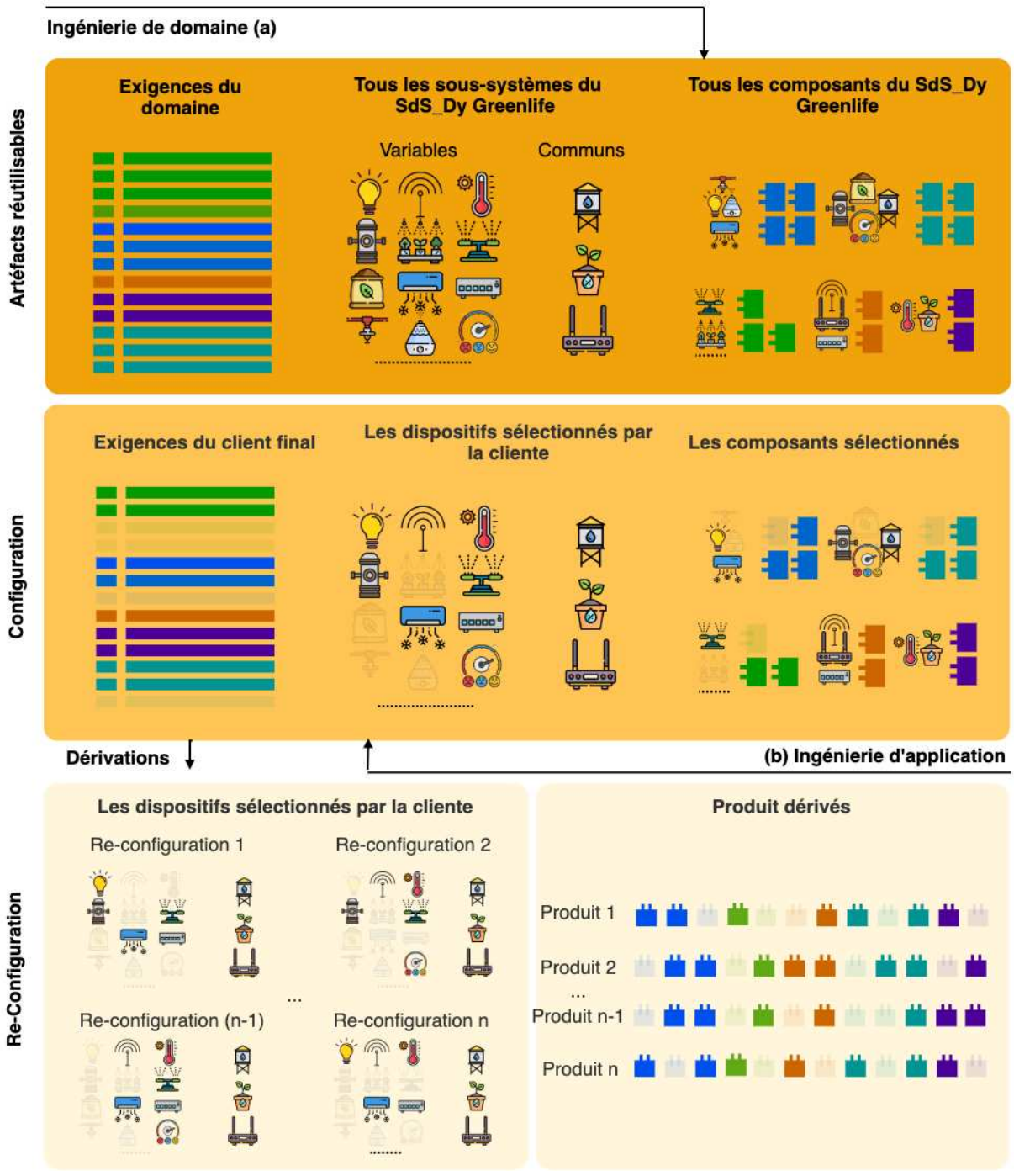

Figure 4-3: Processus de dérivation selon le processus d'ingénierie des LdPD

4.4. Périmètres des systèmes de systèmes dynamiques

Les systèmes de systèmes sont composés de sous-systèmes hétérogènes, collaboratifs et indépendants, qui, ensemble, atteignent un objectif donné (INCOSE 2018). Il est donc important de considérer les exigences relatives au SdS, mais aussi, celles propres à chaque système individuel. Par ailleurs, les exigences existent dans un cadre (Pohl 2010). Il en convient naturellement de spécifier, pour chaque niveau d'ingénierie, les dépendances entre ces exigences et le cadre dans lequel elles existent (Perera et al. 2014). En effet, il est nécessaire d'anticiper les changements qui auront des répercussions sur le bon fonctionnement du SdS, et réciproquement, de définir à l'avance comment le système de systèmes est susceptible de modifier sa configuration en vue de mieux s'aligner à son cadre réel. La conception d'un SdS_Dy requiert ainsi la mise en évidence des constituants de trois dimensions, à savoir le *système*, son *contexte* et son *environnement*, et ce, en considérant la variabilité du périmètre qui les sépare ainsi que la nature et le contenu de leurs interactions :

- Le *système* (de systèmes) est l'arrangement de parties ou d'éléments qui, ensemble, présentent un comportement ou une fonction (INCOSE 2018). Il s'agit ici du système de systèmes dynamiques, en l'occurrence, les systèmes qui le définissent, leurs composants et leurs configurations.
- L'*environnement* est une dimension qui représente « tout » ce qui entoure un système, i.e., l'entreprise, les utilisateurs, les règeles métier, l'environnement physique, les autres systèmes, etc. Il engloble le contexte –définit dans le point suivant, mais aussi, toutes les autres informations qui ne sont pas affectées par le système et n'ont pas d'impact direct sur celui-ci. Il est à noter que certaines de ces informations ont le potentiel d'être de valeur pour le système, dans des situations données (Pohl 2010).
- Le *contexte* est toute information de l'environnement, pouvant être utilisée pour caractériser la situation d'une entité. Dès lors, tout ce qui entoure le système, et qui a un impact direct sur la satisfaction des exigences le concernant fait partie du contexte (Abowd et al. 1999)

Il convient de mentionner que ces dimensions ne sont pas statiques ; des composants du système peuvent faire partie du contexte, et inversement, et des données qui n'ont pas d'impact sur le système, et qui font initialement partie de l'environnement, peuvent à terme influencer son fonctionnement, commutant ainsi au contexte.

4.5. Adaptation du Framework des LdPD aux systèmes de systèmes dynamiques

La spécification des SdS_Dy revient à répondre à des exigences de différentes natures, tout particulièrement, des exigences de variabilité dynamique à plusieurs niveaux d'abstraction. Tout compte fait, considérer un SdS_Dy comme étant une ligne de produits dynamique, capable de dériver les produits qui répondent aux divers besoins des utilisateurs, en tenant compte des altérations provenant de l'entourage, procure les mécanismes nécessaires pour gérer cette variabilité. Ainsi, et conformément au processus d'ingénierie des LdPD, la mise en place d'un SdS_Dy de bout en bout, doit suivre le processus d'ingénierie de domaine, puis d'application, puis d'adaptation (Raúl Mazo 2018).

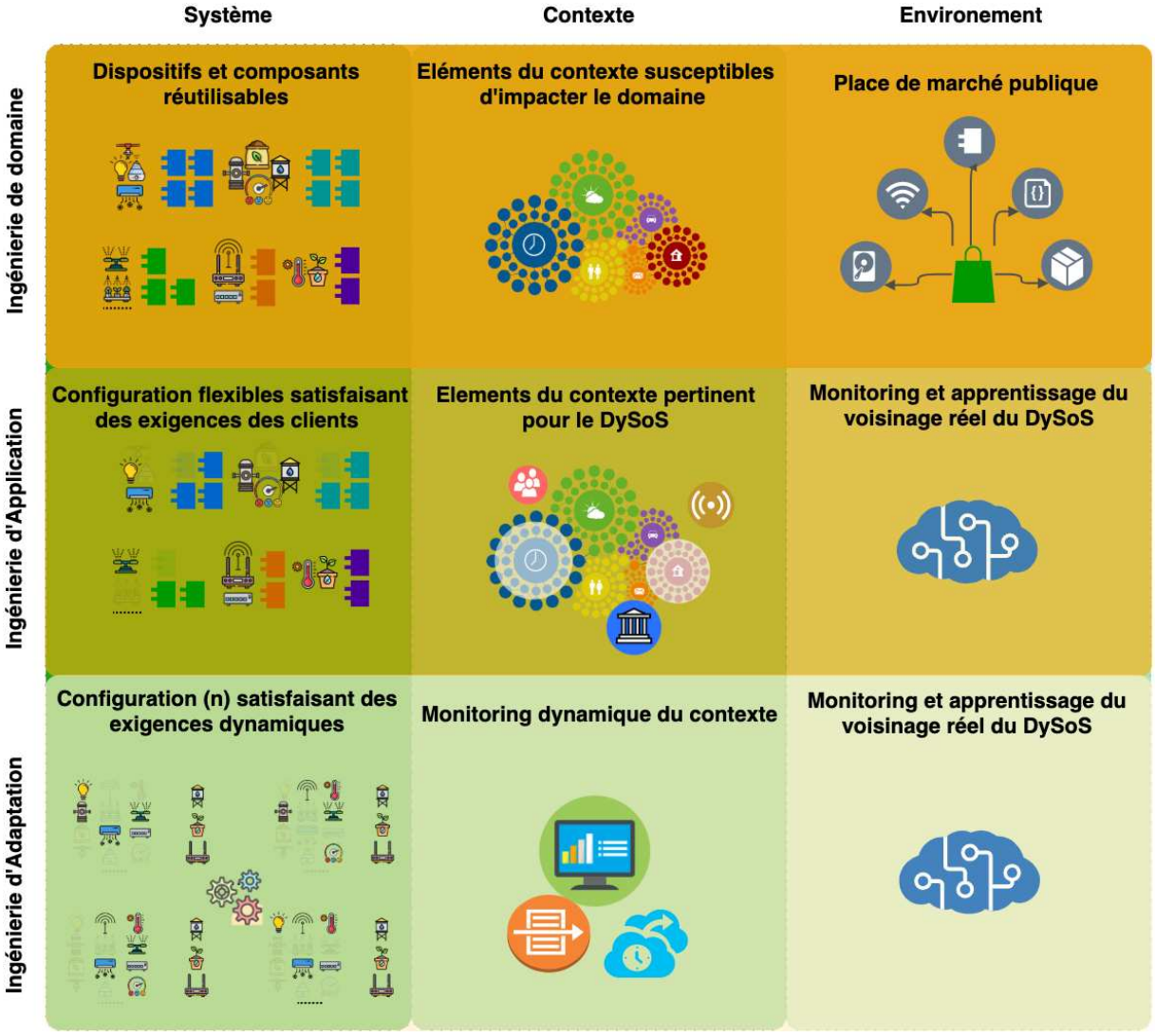

Figure 4-4 : Le Framework SPL4DsoS

Par ailleurs, afin de s'acquitter des spécificités des SdS_Dy, une séparation explicite entre le système, le contexte et l'environnement, en délimitant leurs frontières, et en définissant la nature de leur interfaçage, s'avère nécessaire. La Figure 4-4 présente le cadre des LdPDs, adapté aux SdS_Dy, et adopté dans cette thèse. Ce cadre est désigné "Software Product Lines Framework for (4) Dynamic Systems of Systems" (SPL4DSoS).

4.5.1. Ingénierie de domaine : le système

L'expert du domaine concerné par le SdS_Dy élabore les fonctionnalités que celui-ci est capable de fournir, ainsi que les qualités qui doivent être satisfaites. Cette activité résulte en un cahier des charges qui répertorie les exigences du domaine, qui seront réalisées par l'ensemble des éléments réutilisables, dont certains sont communs à tous les SdS, et d'autres sont spécifiques à des contextes précis. Les termes élément ou système sont généralement employés pour faire référence à un dispositif, un logiciel ou un composant, qui soit pertinent pour des parties prenantes ou des concepts donnés (Sillitto et al. 2019).

Exemple 4.7 : Un compte-gouttes, des sprinklers, des ampoules, des climatiseurs et des fenêtres intelligentes sont des éléments réutilisables pour la création d'une variété de SdS_Dy dans le domaine de l'irrigation. Similairement, pour une ampoule intelligente, des composants réalisés respectivement pour des systèmes d'exploitation TinyOS, Contiki, et Android, sont disponibles.

4.5.2. Ingénierie de domaine : le contexte

La définition du contexte du SdS_Dy pour un domaine donné revient à spécifier les éléments du contexte qui sont susceptibles d'avoir une influence sur les SdS dérivés. Ces éléments peuvent provenir de diverses sources, du fait que le contexte lui-même est défini par ses dimensions interne, externe et temporelle (Schilit et al. 1994). La première dimension regroupe les informations provenant du SdS_Dy. La deuxième dimension fait référence aux aspects externes au SdS_Dy, collectés par le biais de capteurs. Quant au contexte temporel, il se rapporte à un ordonnancement, à des créneaux horaires ou à des instants ponctuels, qui nécessitent une prise en charge particulière. Dès lors, l'analyse du contexte au niveau du domaine permet de cerner tous les événements susceptibles de contrevenir à une performance ultime du produit fini, afin d'acheminer la sélection des composants du SdS_Dy au fur et à mesure que le contexte change.

Exemple 4.8 : Le contexte interne, externe et temporel d'un SdS_Dy peut faire référence, respectivement, à l'état de batterie de certains dispositifs, à la température ambiante ou à une heure de la journée.

4.5.3. Ingénierie de domaine : l'environnement

Au niveau du domaine, l'environnement correspond à une place de marché. Il contient toutes les informations qui se rapportent au domaine : dispositifs, spécifications, exigences de domaine, lois, etc. Cette connaissance de l'environnement est mise à disposition d'un large public et peut contribuer à l'extensibilité dynamique des SdS_Dy. Les services appartenant à un même domaine de connaissance et qui sont nouvellement ajoutés dans cette dimension, sont potentiellement capables de mettre à jour la dimension système en modifiant automatiquement l'ensemble des artéfacts disponibles.

Exemple 4.9 : Un protocole de communication a récemment pénétré le marché télécom. Il peut dorénavant être distribué et utilisé par des ingénieurs ou même des utilisateurs souhaitant mettre à jour leur SdS à partir de la place de marché publique.

4.5.4. Ingénierie d'application : le système

Le processus de configuration démarre par l'identification des exigences spécifiques du client. Elles correspondent normalement à des exigences réutilisables élucidées lors de l'étude de domaine ('idéal étant qu'elles en soient même un sous-ensemble), et -parfois- complétée par certaines plus spécifiques (ce qui s'avère inévitable dans la réalité que nous avons observée). Ainsi, les éléments connexes à ces exigences sont sélectionnés, et le choix des composants se fait sur cette base. Ces derniers sont testés puis déployés afin de dériver un produit final sur mesure. Dans le cas de systèmes auto-adaptatifs, le système généré dans cette dimension est toujours flexible et peut faire l'objet de futures modifications pour générer une application finale, adaptée non seulement aux exigences des utilisateurs, mais aussi au contexte spécifique du SdS_Dy.

Exemple 4.10 : L'utilisateur Maria préfère une irrigation avec Sprinkler. Le compte-gouttes est utilisé exceptionnellement, en cas de surconsommation d'eau. Par ailleurs, au Maroc, où les températures sont relativement élevées, Maria opte pour des ampoules intelligentes, à basse consommation, pour maintenir une température appropriée.

4.5.5. Ingénierie d'application : le contexte

Pour chaque produit dérivé du SdS_Dy, il existe des informations dans son contexte, qui peuvent avoir une influence sur son fonctionnement. Le contexte d'application peut ainsi dépendre de l'environnement d'exécution, comme il peut provenir de la nature de son utilisation et des exigences de ses utilisateurs. Certains éléments du contexte d'applications sont d'ores et déjà recensés lors de l'étude de domaine, mais d'autre aspects sont exclusifs à l'application en question. Cette dimension du Framework fait la part entre les deux, en identifiant ce qui affecte réellement le produit dérivé, tout en rajoutant, parfois, les éléments qui se manifestent après l'exécution du SdS_Dy.

Exemple 4.11: Dans le domaine de l'irrigation, pendant certains jours, il est interdit d'irriguer dans certaines régions pour des raisons écologiques. Ce n'est cependant pas le cas de la région de Doukkala. Quant au contexte «jour de la semaine », il n'est pas considéré pertinent pour le SdS_Dy de Maria et n'est donc pas surveillé. Par ailleurs, l'infrastructure de la région étant instable, il est important de contrôler le courant électrique, ainsi que la connectivité des dispositifs.

4.5.6. Ingénierie d'adaptation : le système

Le processus de reconfiguration est au cœur de cette dimension. Il s'effectue en fonction des résultats de l'ingénierie d'application d'une part et du contexte courant d'autre part. Il est ainsi déclenché à chaque changement significatif du contexte/environnement ou lorsque les exigences d'application sont modifiées.

Exemple 4.12 : Lorsque la consommation d'eau franchit la limite définie par Maria, le système est adapté en conséquence et utilise le Dripline pour irriguer le champ.

4.5.7. Ingénierie d'adaptation : le contexte

Pour chaque produit, le contexte—dont les éléments pertinents sont déclarés dans la dimension du contexte relative à l'ingénierie d'application (section 4.5.5)—est dynamiquement surveillé. Toutes les informations susceptibles de compromettre les performances d'un produit sont collectées et analysées. Toutefois si une inconsistance est détectée, la planification d'un nouveau produit est lancée.

Exemple 4.13: L'humidité du sol est inspectée et rapportée en temps réel, afin de déclencher ou planifier les cycles d'irrigation.

4.5.8. Ingénierie d'application et d'adaptation : L'environnement

L'environnement fait référence à une base de connaissances qui englobe des informations sur le voisinage du produit, des dispositifs qui soient indépendants du SdS_Dy mais qui existent dans son périmètre, des lois locales, des prévisions, etc. Les constituants de cette base ne sont pas forcément monitorés en temps réel, mais demeurent potentiellement significatifs. De plus, le comportement et les habitudes des utilisateurs du SdS_Dy sont aussi surveillés pour affiner les exigences, et accompagner l'évolution des besoins des utilisateurs. Ainsi, l'implémentation de cette dimension permet au SdS_Dy d'assimiler ces changements au fur et à mesure, et par la même, mettre à jour l'instance des exigences propres à cet utilisateur, afin de dériver un nouveau produit plus adéquat.

Exemple 4.14 : La présence d'une personne dans le voisinage du terrain n'est point une information pertinente pour le bon fonctionnement d'un SdS_Dy d'irrigation. Cependant, en cas d'urgence extrême qui requière une intervention surplace, le voisin de Maria peut être contacté à sa place. En outre, parmi les 5 ampoules actives, Maria en éteint très souvent deux à chaque fois que celles-ci sont actives. Ce comportement est assimilé par le moniteur et les exigences de Maria sont modifiées de sorte que pas plus de 2 ampoules soient actives à la fois.

4.6. Conclusion

L'analyse du cahier des charges d'un SdS_Dy d'irrigation d'un champ d'agriculture, puis complétées par une étude systématique de la littérature, révèle plusieurs types d'exigences, dont principalement des exigences nommées ici « exigences de variabilité ». Ce type d'exigences représente la capacité des systèmes de systèmes à être paramétrés, reconfigurés, personnalisés et étendus pour répondre à des besoins et un contexte de fonctionnement évolutifs. Survenant à plusieurs niveaux, il devient ainsi nécessaire de définir les types de variabilité selon 3 axes :

- *Statique et dynamique* : la première étant concernée par le cycle de conception et permet ainsi la génération d'un système valide à partir d'une famille de systèmes,

tandis que l'autre s'étend après l'exécution et génère systématiquement des systèmes, tous valides dans des contextes donnés.

- *Dans le temps et dans l'espace* : les deux font référence à des artéfacts qui existent dans différentes versions. Contrairement à la variabilité dans l'espace qui stipule que toutes ces versions sont valides simultanément, la variabilité dans le temps contraint leur validité à un moment précis.
- *Externe et Interne* : elle permet respectivement à l'utilisateur et aux concepteurs de choisir entre plusieurs possibilités de variantes pour des fins distinctes.

Dès lors, dans ce chapitre, le cadre SPL4DSoS est présenté. Il puise ses concepts fondamentaux de l'ingénierie des lignes de produits logiciels afin de définir les artéfacts variables dans la spécification de SdS_Dy. Par ailleurs, il incorpore les bonnes pratiques d'ingénierie des exigences, en intégrant des dimension verticales délimitant le système, son contexte et son environnement. Ceci permet de spécifier la dynamique entre les exigences à chaque niveau d'ingénierie (domaine, application et adaptation) avec le contexte d'influence d'une part, et l'environnement neutre mais pertinent d'autre part.

Chapitre 5 Aperçu du langage de spécification de systèmes des systèmes dynamiques proposé

5.1. Introduction

La spécification efficace de la variabilité du système, du contexte et de l'environnement, est l'objectif principal de l'étude de domaine, d'application et d'adaptation. Cependant, la multitude de vues résultantes de cette distinction, représentées dans le cadre SPL4DSoS, ainsi que les différents points de vue d'un SdS_Dy qui sont concernés par la spécification, compromettent la cohérence des exigences de bout en bout. Ce chapitre introduit les notions fondamentales pour l'élaboration d'un langage pour la spécification formelle des exigences spécifiques à cette catégorie de systèmes. Le chapitre présente le concept de séparation des préoccupations, en s'appuyant sur l'ingénierie dirigée par les modèles. Ce principe est mis en œuvre par différents modèles soutenant la spécification des perspectives des SdS_Dy, tout en garantissant une consistance au long du processus de conception, de configuration et d'exécution. Ensuite, le chapitre rappelle les objectifs principaux, avant d'introduire par la suite les modèles implémentés.

5.2. Vue d'ensemble des modèles du Framework SPL4DSoS

La modélisation efficace et flexible d'un SdS_Dy, dans le cadre du Framework SPL4DSoS proposé dans la Section 4.5, peut s'appuyer sur la pratique de séparation des préoccupations pour modéliser les aspects spécifiques au système, au contexte et à l'environnement. L'objectif de cette section qui suit, est de passer en revue les modèles proposés pour modéliser ces dimensions. La Figure 5-1 résume l'ensemble de ces modèles, impliqués dans les différentes phases d'ingénierie, à savoir l'ingénierie de domaine, d'application et d'adaptation.

- **Le Modèle de Variabilité (MV)** est concerné par l'étude des exigences réutilisables, ainsi que les artéfacts possibles pour la création d'un système dans le cadre de l'ingénierie de domaine. De ce point de vue, il est question de spécifier tous les éléments susceptibles de constituer un système final et de définir les règles de dépendances (Sawyer et al. 2012b). En fonction de la perspective de l'approche, des caractéristiques, des points de variation ou des exigences—entre autres—sont au cœur du modèle de variabilité. Des approches comme (Batory 2005) (Schneeweiss and Hofstedt 2013) ou (Quinton et al. 2013) utilisent des variations du modèle de caractéristiques, basées sur la notation FODA. Dans ces variations de modèles, tous les

produits d'un LdP sont représentés dans un modèle compact en utilisant la notion de caractéristique (feature en anglais), pour abstraire, hiérarchiser et contraindre toute partie physique, logique ou comportemental du système. D'autres travaux, par exemple ceux de Dhungana et al. (Dhungana et al. 2011), se basent sur des modèles de décision pour modéliser la variabilité, en définissant les points de variation, ainsi que l'ensemble des choix disponibles à un moment donné au cours de la dérivation d'un produit. Finalement, Semmak et al. (Semmak et al. 2010) modélisent la variabilité des exigences dans un modèle orienté buts, cette première étant intrinsèquement reliée aux caractéristiques d'un SPL.

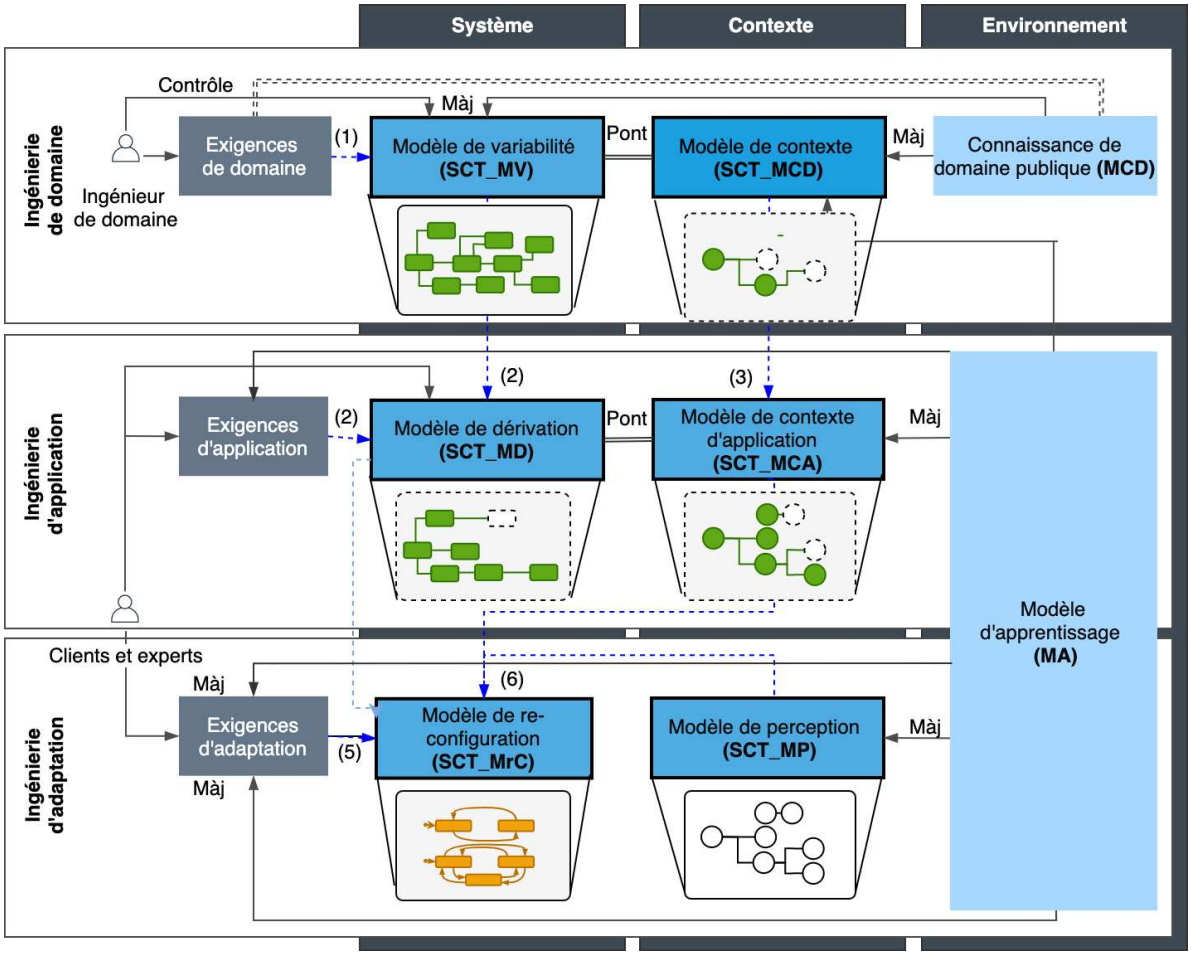

Figure 5-1 : Projection du langage SCT dans le Framework SPL4DSoS

- **Le Modèle de Contexte de Domaine (MCD)** est implanté par le biais d'un modèle de contexte qui définit de manière formelle (ou semi-formelle) les informations qui entourent la LdP et qui sont susceptibles d'avoir un impact sur celle-ci.
- **Le Modèle de Connaissance Publique (MCP)** correspond à la vue qui englobe tous les artéfacts qui appartiennent au domaine de connaissances concerné par la LdP.

Contrairement au modèle de variabilité, qui est aussi responsable d'organiser cette connaissance sous formes d'artefacts réutilisables, un modèle de connaissance va au-delà des services pertinents pour l'organisme en question et disponible « in-house », car il regroupe, organise et spécifie cette connaissance de manière générique, indépendante, accessible au public et fréquemment mise à jour (Kumar et al. 2010). Les composant de domaine ont les caractéristiques listées dans (Raul Mazo 2018; Sommerville 2015). Dans le domaine des LdP, les modèles de connaissances sont souvent mentionnés pour garantir l'évolutivité d'une ligne de produits (Hotz et al. 2003; Krebs et al. 2002), qu'il s'agisse de l'évolution des artéfacts réutilisables, ou des exigences du domaine. Par exemple, Tomer et Schach dans (Tomer and Schach 2000) utilisent un arbre d'évolution pour « maintenir » la ligne de produit. Il s'agit d'un modèle à deux dimensions, qui d'un côté, assure le suivi du développement de produits individuels et d'un autre côté, assure la maintenance de la LdP en modifiant des artefacts à la suite de modification d'exigences. Les modèles de connaissances sont rarement implémentés en pratique. Cependant, le contrôle de cohérence qui suit la modification d'un modèle de variabilité, souvent dans des modèles de contrôle de cohérence dédiés à cet effet, fait l'objet de plusieurs travaux (Weyns and Michalik 2011; Quinton et al. 2014; Benlarabi 2014).

- **Le Modèle de Configuration (MC)** est relié d'abord aux exigences d'application, ensuite au modèle de variabilité —étant une dérivation flexible de celui-ci—et finalement au contexte d'application pertinent pour l'utilisateur. En effet, les exigences des clients du produit final, qui sont décrites en langage naturel, ou dans des diagrammes semi-formels tels que les modèles d'exigences, font l'objet d'une mise en correspondance avec les exigences du domaine, afin de sélectionner les éléments du système qui satisfont ces premières. De surcroit, le contexte rajoute une série de contraintes aux éléments à sélectionner, selon une logique prédéfinie.
- **Le Modèle de Contexte d'Application (MCA)** est un sous-ensemble de DCM qui décrit les éléments du contexte ayant un impact sur le choix des éléments du système. Ce modèle est généré sur la base des exigences des clients finaux. La plupart des approches maintient l'usage des modèles de caractéristiques (Raúl Mazo 2014), parfois étendus (Fernandes et al. 2008; Pfannemueller et al. 2017), pour spécifier le contexte. D'autres notations se basent sur une définition ontologique des éléments du contexte (Jaroucheh et al. 2010). Certains auteurs, proposent des modèles

complémentaires aux modèles de variabilité, comme des modèles de transition (Bencomo et al. 2008) ou des modèles organisationnels (Peña et al. 2011), pour documenter la variabilité liée au contexte. La gestion des dépendances entre un modèle de contexte et un modèle de variabilité, est parfois intégrée à ce premier et elle fait l'objet parfois d'un modèle intermédiaire de raisonnement. Celui-ci détermine l'effet d'une instance du contexte, sur la ligne de produits.

- **Le Modèle de re-Configuration (MrC)** est au cœur du système auto-adaptatif. En effet c'est à ce niveau que les règles d'assemblage et les contraintes d'adaptation sont définies. Le système flexible dérivé au niveau de l'ingénierie d'application, subit de nouvelles restrictions pour générer un système final adapté dans un contexte spécifique. Cette analyse est faite par le biais de modèles de reconfiguration, souvent implantés sous forme de langage dédié (Voelter and Groher 2007; Wolter et al. 2006; Asikainen et al. 2004) et supporte la dérivation d'un modèle de variabilité résolu.
- **Le Modèle de Perception (MP)** dispose de mécanismes nécessaires pour observer le contexte, et tel qu'il est défini par le modèle de contexte d'application. Les données appréhendées sont ensuite véhiculées au modèle de reconfiguration selon une logique prédéfinie par le biais d'un modèle de perception. Celui-ci fait généralement référence à des observateurs (Parra et al. 2011) ou des simulateurs (Morin et al. 2009).
- **Le Modèle d'Apprentissage (MA)** est un modèle cognitif qui observe le comportement de l'utilisateur et l'évolution du contexte. Il communique aux modèles associés les nouvelles habitudes et exigences ou contextes pertinents. En effet, le comportement de l'utilisateur est comparé aux exigences d'application, les écarts existants sont analysés et le modèle de configuration est modifié en conséquence. A notre connaissance, le modèle d'apprentissage n'est spécifié dans aucun des langages des LdP étudiés. De la même manière, les changements de contexte qui ont un impact sur la LdP et qui n'ont pas été anticipés par le modèle de contexte d'application, peuvent dynamiquement être intégrés dans ce dernier, tout comme les dépendances avec le modèle de variabilité qui en résultent, au moyen de techniques comme l'apprentissage automatique (machine Learning en anglais) (Sharifloo et al. 2016).

Les modèles introduits dans cette section présentent une vue globale des points de vue qui rentrent en jeu dans la spécification de SdS_Dy. Selon les besoins et préférences des ingénieurs, des modèles peuvent être implémentés dans des paradigmes adéquats, tout en respectant la logique et les dépendances impliquées.

5.3. Positionnement des contributions

L'étude des SdS_Dy conjuguée à l'étude de cas GreenLife permet de mettre en évidence les caractéristiques suivantes :

- Les systèmes impliqués dans un SdS_Dy peuvent opérer individuellement. Ils peuvent également fonctionner en collaboration avec d'autres systèmes. En conséquence, ils peuvent faire l'objet d'une réadaptation ponctuelle ou se reconfigurer collectivement.
- Des systèmes peuvent appartenir, simultanément et consécutivement à plusieurs SdS_Dy. Dès lors, les exigences qu'ils satisfont sont variables et les configurations qui en résultent sont dynamiques afin de répondre aux besoins parfois hétérogènes des différentes SdS et utilisateurs qu'ils desservissent.
- Des standards, des protocoles, des langages et des spécifications, entre autres, s'établissent régulièrement. Il résulte de cela que la fréquence d'évolution des SdS_Dy dans le temps est considérable.
- Les besoins en reconfiguration peuvent provenir de changements dans le contexte des SdS_Dy, à un instant particulier d'un changement dans la configuration ou dans la composition d'un sous-ensemble de systèmes.

Sur la base de ces caractéristiques est établi le besoin d'un langage de spécification formelle de SdS_Dy. Ce langage de spécification doit être conforme à la vision d'ingénierie des lignes de produits logiciels, introduit dans le Framework SPL4SoS. Il convient de mentionner qu'on suppose que les exigences sont d'ores et déjà élucidées. Les travaux de Sousa et al. (Sousa et al. 2019) et de Alawairdhi et al. (Alawairdhi and Aleisa 2011) peuvent servir de base pour cette phase du processus d'ingénierie d'exigences selon l'approche des LdPD.

En accord avec les standards d'ingénierie des systèmes et du logiciel proposés dans (ISO 2017) et adapté dans (Silva et al. 2019), le langage doit permettre de réaliser les objectifs suivants, sont aussi illustrés dans la Figure 5-1.

(1) Définir les éléments pertinents au domaine, ainsi que leurs dépendances.

(2) Définir les exigences des utilisateurs, sur lesquelles s'opèrent les choix nécessaires à une dérivation d'un modèle de configuration.

(3) Définir sur la base de (2) un modèle de configuration.

(4) Définir les éléments du contexte qui sont pertinents pour le SdS_Dy, et ce dépendamment de son exécution et utilisation. Certains de ces éléments sont relatifs aux choix et décisions définis dans (2).

(5) Définir les exigences d'adaptation, sur lesquelles s'appuient les choix nécessaires à une reconfiguration d'une application dérivée d'un modèle de configuration

(6) Définir l'impact de (5), sur la dérivation d'une configuration valide, sur la base du modèle de configuration(MC), pour une instance du modèle de contexte d'application (MCA), représenté par le modèle de perception (MP).

Pour conclure, la contribution présentée dans cette thèse présente un langage de spécification formelle qui respecte et appuie les particularités des SdS_Dy, notamment en termes de variabilité et de dynamique, et qui génère en sortie des configurations pertinentes dans des contextes donnés.

5.4. Affinement des contributions

Nos contributions ***Contr3***, ***Contr4*** et ***Contr5***, énoncées dans l'introduction de cette thèse, font référence aux artéfacts suivants :

Contr 3. le langage de spécification formelle des systèmes de systèmes dynamiques,

Contr 4. le processus de configuration,

Contr 5. les outils exploités et mis en œuvre pour faciliter ces deux premiers.

Nous précisons ces aspects ci-dessous. Par ailleurs, afin de mieux cerner les explications présentées dans les chapitres suivants, il convient de distinguer la terminologie suivante :

- **Le langage SCT** fait référence à l'ensemble des concepts qui permettent de créer des modèles selon les règles syntaxiques et sémantiques définies. C'est un langage formel qui représente des états, des transitions, et des contraintes, et ce dans une machine à états finis. Il peut être réalisé sous format graphique ou textuel. Dans sa version courante, seule la version textuelle est implémentée.
- **Le modèle élaboré selon la notation SCT** (parfois appelé modèle SCT) présente une instance du méta-modèle SCT, introduit dans le chapitre 6. L'interface pour l'élaboration de ce modèle est unique, cependant, un modèle SCT intègre implicitement une projection sur cinq modèles différents du Framework SPL4DSoS, à savoir le modèle de variabilité de domaine, le modèle de configuration, le modèle de contexte d'application, le modèle de reconfiguration, et le modèle de perception.

- **Le processus de spécification selon la notation SCT** présente les étapes suivies par le réalisateur du modèle selon la notation SCT. Il consiste en la création de la machine à états finis, la définition du système ainsi que l'environnement qui l'impact sous forme de variables, la spécification des exigences de domaine sous forme de contraintes, la définition des états de chaque élément ou sous-ensemble d'éléments du SdS_Dy, et finalement, au regroupement des exigences qui doivent être satisfaites dans un état particulier, en les traduisant en contraintes.

5.4.1. Aperçu du langage de spécification (Contr3)

Étant au cœur de la 3ème contribution (***Contr3)***, le langage SCT a pour vocation de spécifier formellement des SdS_Dy. Pour atteindre cet objectif, le langage doit fournir les outils nécessaires pour gérer la variabilité des exigences qui se manifeste à plusieurs niveaux.

L'implémentation courante du langage SCT est basée sur 5 modèles du cadre SPL4DSoS, du fait qu'il permette en premier temps de spécifier des exigences de domaine dans un modèle de variabilité (SCT_MV), ensuite les exigences d'application ainsi que le contexte qui l'accompagne respectivement dans un modèle de configuration (SCT_MC) et de contexte d'application (SCT_MCA), et finalement les exigences dynamiques dans un modèle de reconfiguration (SCT_MrC) accompagné de modèle de perception (SCT_MP) pour informer sur le contexte réel du SdS_Dy. Les différents modèles impliqués dans le langages SCT sont des projections de SPL4DSoS, tel qu'illustré dans la Figure 5-1.

- **Une projection sur le modèle de variabilité (SCT_MV)** afin de produire un modèle de variabilité, en spécifiant les éléments du SdS_Dy, ainsi que leurs relations, et ce sous forme de contraintes.
- **Une projection sur le modèle de contexte de domaine (SCT_MCD)** dans le but de spécifier les variables de contexte anticipée par les ingénieurs de domaine
- **Une projection sur le modèle de configuration (SCT_MC)** pour intégrer les règles d'adaptabilité dans le modèle d'application. Celui-ci peut également inclure les évolutions par rapport au modèle de variabilité de domaine (MV), et ce, qu'il s'agissent d'évolution d'exigences, ou des composants du SdS_Dy lui-même.
- **Une projection sur le modèle de contexte d'application (SCT_MCA)** pour définir les éléments du contexte qui ont un impact sur les performances ou la pertinence de la configuration du SdS_Dy.

- **Une projection sur le modèle de reconfiguration (SCT_MrC)** afin de définir les états de configurations, ainsi que les exigences qui sont satisfaites lorsque les états respectifs sont actifs.
- **Une projection sur le modèle de perception (SCT_MP)** pour connecter le programme de contraintes généré au contexte réel, afin de dériver dynamiquement des produits valides.

Tout compte fait, l'ingénierie dirigée par des modèles assure et facilite la conception de systèmes traitant une multiplicité de points de vue. Néanmoins, un souci d'hétérogénéité émerge. Afin d'appréhender la cohérence des exigences spécifiées à travers l'ensemble des modèles du langage, la transformation des concepts vers un langage uniforme devient pertinente. Celui-ci doit par ailleurs disposer de capacités d'analyse, afin de permettre la génération de configurations valides.

5.4.2. Aperçu du processus de configuration des SdS_Dy (Contr4)

Une configuration, dans le domaine des lignes de produits logiciels, fait référence à un produit valide dérivé de la ligne de produits. Dans le contexte des SdS_Dy, ce concept correspond à une configuration du système de systèmes, et ce sur la base d'éléments de configuration réutilisables. Le processus de configuration, concerné par la contribution ***Contr4***, correspond à la méthode utilisée pour fournir un support automatisé des différentes taches impliquées dans la dérivation d'un produit valide, sur la base des exigences des différents processus d'ingénierie. Une variété d'approches de configuration existe à cette fin (Benavides et al. 2010). Dans cette thèse, et sur la base des travaux réalisés dans (Salinesi et al. 2011; Raúl Mazo et al. 2012; Sawyer et al. 2012a; Djebbi, Salinesi, and Diaz 2007; Muñoz-Fernández et al. 2014; Benavides, Trinidad, and Cortés 2005), nous affirmons que la spécification d'une LdPD peut être réduite à un problème de satisfaction de contraintes (PdSC) (Constraint Satisfaction Problem : CSP en anglais). Autrement dit, que tous les modèles nécessaires pour la spécification d'une LdPD peuvent être transformés en un programme de contraintes, qui, une fois résolu, présente une configuration valide. Celle-ci prend en considération le contexte courant en communiquant avec le modèle de perception, ainsi que les exigences mises en vigueur.

5.4.3. Aperçu du langage SCT (Contr5)

Les modèles mis en évidence dans la Figure 5-1, par des bordures noires, sont implémentés dans le langage SCT en se basant sur les capacités offertes par le Framework Xtext (Bettini 2013), faisant ainsi l'objet de la contribution ***Contr5*** D'un côté la syntaxe est définie, afin d'exprimer les règles de réalisation de modèles SCT. Et d'un autre, la sémantique est assurée en traduisant les concepts du modèle SCT en variables et en contraintes appliquées sur ces variables, dans un problème de contraintes.
Le langage se focalise sur les aspects liés à la spécification des exigences des SdS_Dy, et la transformation des modèles impliqués dans cette spécification sous forme de programme de contraintes. Bien que le programme de contraintes généré permette de réaliser certaines opérations telles que la vérification et la simulation, ces deux derniers aspects sont hors périmètre de cette thèse.

5.5. Conclusion

L'ingénierie des lignes de produits logiciels est une approche d'ingénierie dirigée par des modèles, et ce principalement, afin de séparer les préoccupations liées à la spécification des différents points de vue d'une ligne de produits. Les SdS_Dy ne font pas l'exception. Effectivement, on distingue une multitude de modèles intervenants dans la mise en pratique du Framework SPL4DSoS, ceux-ci correspondent à des modèles de :

- Variabilité de domaine et d'application, pour déterminer les éléments communs et variables des SdS_Dy, ainsi que leurs dépendances ;
- Contexte de domaine et d'application, pour formuler une définition rigoureuse des éléments du contexte qui ont un impact sur la performance et sur le comportement du SdS_Dy;
- Connaissance, pour faciliter et accompagner l'évolution de la composition du SdS_Dy d'un côté, et des exigences de domaine d'un autre côté ;
- Configuration, pour dériver un produit conformément aux règles définies par le modèle de variabilité et aux exigences d'application dans un contexte donné ;
- Perception, pour surveiller les éléments du contexte en temps réel et notifier des évolutions significatives en concordance avec le modèle de contexte ;
- Apprentissage, pour récupérer le feedback du système par rapport au comportement de l'utilisateur d'un côté et par rapport à des contextes divers d'un autre côté, pour

planifier des mises à jour dans les modèles d'exigences d'application et les modèles de contexte.

Dans cette thèse, le langage pour la spécification de SdS_Dy est établi. Il implémente spécifiquement des modèles de variabilité, de contexte, de configuration et de perception. Ceux-ci permettent de définir des modèles de variabilité flexibles et évolutifs ainsi que les règles qui régissent le comportement du SdS_Dy, et ce, sur la base de la nature et état du contexte. Ainsi, des groupes d'exigences sont définies sur un ensemble d'éléments du SdS_Dy. La mise en vigueur de chacun de ces groupes est gérée par le modèle de contexte. Au bout du compte, tous ces modèles sont traduits en contraintes, générant de la sorte, un programme de satisfaction de contraintes, dont la(les) solution(s) représente(nt) une (des) configuration(s) valide(s) du SdS_Dy.

Partie III – Contributions

Chapitre 6 Le graphe États-Contraintes Transitions (SCT)

6.1. Introduction

Les machines à états finis (Finite State Machine : FSM en Anglais) (Harel 1987) sont utilisées comme outil de spécification de systèmes réactifs. Harel propose une sémantique formelle à la fois simple et puissante dans laquelle les états décrivent les caractéristiques opérationnelles d'exécution du système et les transitions décrivent les événements qui déclenchent le passage d'un état à un autre. Depuis lors, le formalise des machines à états finis a été sans cesse enrichi de nouveaux concepts permettant de gérer différents aspects des systèmes réactifs ; on peut en particulier retenir celles destinées à la spécification de systèmes auto-adaptatifs. Ainsi, Neto et al. (Neto et al.1999) combinent des états avec des hypothèses de domaine, pour définir les adaptations conformes aux exigences requises. Tesei et al (Tesei et al. 2013) utilisent deux machines à états distinctes : la première pour décrire le comportement réel du système, l'autre visant à décrire les contraintes environnementales qui déclenchant les adaptations du systèmes spécifié. Enfin, (E. Lee et al. 2018) propose d'utiliser deux machines à états (SA-FSM et A-FSM) pour la définition concrète et abstraite du système auto-adaptatif, accompagné de processus d'auto-adaptation basé sur une boucle MAPE.

Ces variantes des machines à états destinées à la spécification de systèmes auto-adaptatifs ne sont pas suffisantes pour la spécification de SdS_Dy pour plusieurs raisons : d'une part, elles ne permettent pas de spécifier des états de configuration qui représentent fidèlement les exigences correspondant à l'état. De plus, ces variantes font l'hypothèse qu'une configuration unique est possible dans un état de configuration donné. En d'autres termes, état de configuration égal configuration et non espace de configuration. La conséquence est que la seule manière de représenter les espaces de configuration est au travers de l'énumération d'état, ce qui entraîne une explosion combinatoire du nombre d'états à spécifier, ainsi qu'une complexification artificielle inextricable des modèles.

Les graphes États-Contraintes Transition (SCT), sont une proposition originale de notation pour la spécification formelle de systèmes réactifs qui enrichit le formalisme des machines à états finis avec le concept de contraintes permettant ainsi de spécifier les exigences des Systèmes de Systèmes dynamiques. Ce chapitre commence par donner une définition formelle des concepts sur lesquels se base le langage SCT. Dans un second temps, il décrit la syntaxe et la sémantique de la notation proposée. Enfin, le chapitre conclut en présentant la spécification du système GreenLife.

6.2. Définitions formelles

Quelques concepts fondamentaux utiles à la formalisation des SCT ont été introduits dans les chapitres précédents font référence à l'ingénierie des LdP : les modèles de variabilité, la ligne de produits (statique et dynamique), le produit et la configuration. Ils se rapportent aussi aux problèmes de satisfaction de contraintes. Pour la formalisation des SCT, il convient de garder à l'esprit les définitions formelles suivantes :

Définition (Modèle de variabilité). Un modèle de variabilité est défini à travers le tuplet (V, A, Dom, α, d, D, Card, GCard) par Cordy et al. (Cordy et al. 2013), Lauenroth and Pohl (Lauenroth and Pohl 2007) et (Mazo et al. 2012), tel que :

- $\boldsymbol{V}$ *est un ensemble de variables non vide,*
- $\boldsymbol{A}$ *est l'ensemble des attributs liés à chaque variable,*
- $\boldsymbol{DomValue}$ *est l'ensemble des valeurs de domaines valides, sous la forme Domvalue =* $\{\mathbb{Z} \cup \mathbb{Q} \cup \{0,1\} \cup enum\}$,
- $\boldsymbol{\alpha \in A \rightarrow V}$ *est une fonction qui affecte un attribut à une variante numérique,*
- $\boldsymbol{Dcm: V \times V \rightarrow \{Bool, Arithm, Symb\}}$ *est une fonction qui codifie les dépendances entre les variantes, ou Bool, Arithm et Symb expriment respectivement des relations booléenne, arithmétique ou symbolique entre un ensemble de variables.*
- $\boldsymbol{Card: V <i,j> \rightarrow \mathbb{N} \times (\mathbb{N} \cup \{*\})}$ *détermine la cardinalité de chaque variante, où* $i, j \in \boldsymbol{N}$ *correspondent respectivement au nombre minimal et maximal des instances requises. * spécifie un maximum non défini. Lorsque V est une mono-instance, i=j=1.*
- $\boldsymbol{GCard: (v_1, v_2 \ldots v_n) <i,j> \rightarrow \mathbb{N} \times (\mathbb{N} \cup \{*\})}$ *détermine la cardinalité de groupe d'un groupe de variantes. où* $i, j \in \mathbb{N}$ *correspondent respectivement au nombre minimal et maximal des variables requises du groupe de variables concerné par la cardinalité de groupe. * spécifie un maximum non défini.*

Définition (Ligne de produits - LdP) Une ligne de produits est définie par Lauenroth and Pohl (Lauenroth and Pohl 2007) par l'ensemble des couples **P(P(V), P(Val))** qui représente l'ensemble des produits, tel que :

- $\boldsymbol{P(V)}$ *est un ensemble de variantes,*
- $\boldsymbol{P(Val)}$ *est l'ensemble des évaluations des attributs liés à P(V).*

Définition (Configuration) étant donné un modèle de variabilité défini par (V, A, DomVal, α, d, D, Card, GCard), une configuration une instanciation du modèle de variabilité, où chaque variable est instanciée en respectant **α**. Une configuration est considérée valide par Cordy et al. (Cordy et al. 2013) lorsque :

- *Une variable est évaluée dans Dcm tel que* $\boldsymbol{Dcm: V \times V \rightarrow \{Bool, Arithm, Symb\}}$

- *Un groupe de variables est sélectionné en respectant GCard, c'est-à-dire tel que* ***GCard (E) = {n, m} ⇒ n≤ | E | ≤m. Avec E⊆V***
- *Chaque variable multi-instanciée est sélectionnée en fonction de sa cardinalité telle que* ***Card (v[i])={n,m} ⇒ n≤ |F| ≤m.Where F⊆{v[1], … v[i]}, and ∀ v[k] ∈E, v[k]>0, with k ∈ {1,i}***
- *Les attributs liés à chaque variable satisfont des contraintes c, conformément à la définition 8.*

Définition (Lignes de produits dynamiques - LdPD) Une ligne de produits dynamiques est définie par le tuplet (P(V), P(Val), u, p, t) où :

- ***P (V)*** *est un ensemble de variantes,*
- ***P (Val)*** *est l'ensemble des évaluations des attributs liés à P (V), tel que Val: A ⟶ Dom,*
- ***t*** *est un contexte,*
- ***p ⊆P (P (V), P (Val))*** *est un ensemble de produits dérivés de la ligne de produits,*
- ***u :*** *t ⟶ p est une fonction associant chaque contexte à des produits valides.*

Définition (PdSC) Un problème de satisfaction de contraintes est défini dans (Rossi et al. 2006) et (Van Hentenryck 1987) par le triplet (X, D, c), où :

- ***X*** *est un ensemble fini de variables,*
- ***D*** *est un ensemble fini de domaines,*
- ***c*** *est une contrainte définie sur X, par rapport à son domaine D.*

Définition (Machine à automates finis – FSM) Une machine à automates finis est définie dans (Hooman et al. 1988) par le tuplet (SS, CS, r, T, a)) où :

- ***SS*** *est un ensemble d'états simples,*
- ***CS*** *est un ensemble d'états composites,*
- ***r*** *est un ensemble de régions,*
- ***T*** *est un ensemble de transitions et I∈T est la transition initiale,*
- ***a*** *est un ensemble d'actions.*

6.3. Méta-modèle SCT

Le langage de spécification des modèles États-Contraintes Transitions (State-Constraints Transitions, SCT en anglais), est un langage de spécification formelle fondé sur la théorie des automates finis (Harel 1987; Hooman et al. 1988). Les modèles de SCT sont des constructions abstraites qui définissent tous les états du SdS_Dy, ainsi que des transitions qui activent le passage d'un état à un autre état à l'occurrence d'un événement. Les transitions sont spécifiées par l'événement déclencheur et les conditions à rencontrer pour passer de l'état source à l'état cible de la transition. L'événement et les conditions des transitions font référence au contexte, conformément au principe d'adaptation

contextuelle des SdS-dyn. Chaque état est spécifié sous forme de contraintes qui formalisent les exigences que le SdS_Dy doit satisfaire. Comme tout problème de contrainte, la solution à l'ensemble des contraintes associées à un état peut être multiple. La sémantique formelle d'un état de SCT est qu'à chaque instant t, la configuration du SdS-dyn est conforme aux contraintes associées à l'état dans lequel il se trouve. De cette définition découlent deux conséquences :

- D'une part, un SdS-Dyn peut changer de configuration tout en restant dans le même état, dès l'instant que les exigences associées à l'état sont respectées. Cela peut par exemple être le résultat d'une demande explicite de l'utilisateur tel qu'un paramétrage.
- D'autre part, lors d'une transition un SdS-Dy peut changer d'état sans changer de configuration dès lors que cette configuration satisfait à la fois les contraintes de l'état source et de l'état cible de la transition.

Le méta-modèle illustré dans la Figure 6-1 présente une vue globale du langage SCT, en faisant apparaitre les principaux concepts au travers de 3 packages : variabilité, contexte, et configuration.

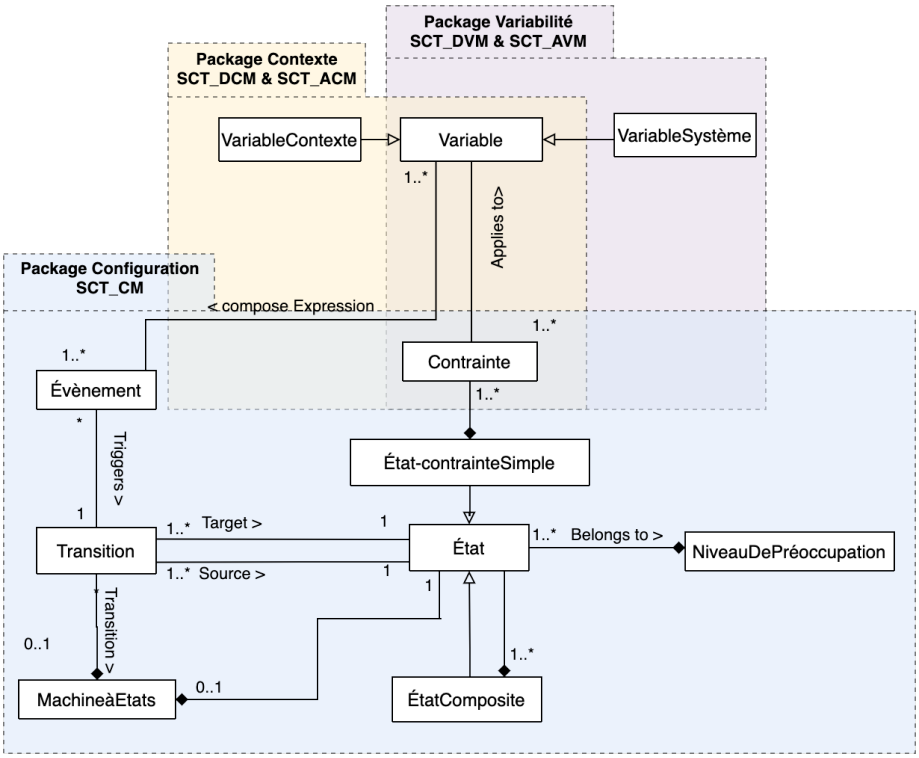

Figure 6-1: Méta-modèle du langage SCT

- **Le package de variabilité** fait référence aux modèles MV et MC qui décrivent respectivement les deux niveaux de variabilité : la variabilité au niveau du domaine, et la variabilité au niveau de l'application où on retrouve aussi les exigences qui peuvent apparaitre et disparaitre en fonction du contexte. Les variables des modèles de variabilité font référence à tous les éléments de la LdPD. Le terme élément est ici utilisé dans un sens large : il peut s'agir de composants logiciels ou matériels, de sous-systèmes ou même de propriétés fonctionnelles ou non fonctionnelles du système de systèmes, etc. Les dépendances entres les éléments de la LdPD sont spécifiées formellement au moyen de contraintes appliquées aux variables du package de variabilité et du package de contexte.
- **Le package de contexte** fait référence au modèle MCA qui définit les aspects de contexte auxquels le SdS-Dyn doit réagir. Le contexte y est défini au moyen de variables qui représentent des abstractions d'une partie de l'environnement de fonctionnement du SdS-Dyn qui devra donc être surveillée à l'exécution. Les variables de contexte peuvent être connectées les unes aux autres par le biais de contraintes qui décrivent la logique de cette relation. Le « contexte » est donc défini comme la combinaison de toutes les variables de contexte.
- **Le package de configuration** fait référence au modèle MrC qui décrit les différents états du SdS-Dyn, qui peuvent être simples ou complexes. Lorsqu'ils sont complexes, les états du SdS-Dyn peuvent être spécifié par composition d'états eux-mêmes simples ou complexes et dont la spécification est nécessaire à une formalisation plus fine du comportement des constituants du SdS-Dyn. Les états simples, dénommés « état-contraintes simples », correspondent aux exigences qui doivent être satisfaites lorsque le SdS est dans l'état concerné. Le statut des états, qui détermine si le comportement imbriqué doit être instauré ou pas, est déterminé par le biais de transitions, porteuses de conditions sur la situation du contexte. Comme le SdS_Dy assemble plusieurs dispositifs qui ont leur propre modèles de domaines, le concept de ConcernLevel, apporte une notion de région, auquel est assigné un groupe d'états conjoints, créant ainsi des états concurrents pour les dispositifs qui coexistent dans le SdS-Dy.

6.4. La syntaxe SCT

La description du langage SCT applique les directives de Harel et Rumpe (Harel and Rumpe 2000) qui suggère d'établir d'abord la syntaxe concrète et abstraite du langage, avant de définir la sémantique de ses concepts. La syntaxe abstraite introduit les concepts utilisés par le langage ainsi que les relations qui régissent leurs liaisons. La syntaxe concrète précise la représentation lisible du langage. Ces aspects sont introduits grâce à une grammaire EBNF (Feynman and Objectives 2016). Ce choix de représentation est motivé par la nature textuelle du langage implémenté, car la grammaire permet de définir explicitement les règles syntaxiques, et implicitement la terminologie employée pour ce faire. Le Tableau 6-1 résume les symboles clés employés pour définir notre grammaire.

Symbole	Description
<non-terminal>	Variables syntaxiques qui définissent le langage
« terminal »	Chaines de caractères qui apparaissent dans le langage
…+	(Non) Terminal qui peut être instancié en une ou plusieurs occurrences
…*	(Non) Terminal qui peut être instancié en zéro ou plusieurs occurrences
… ?	(non) terminal optionnel

Tableau 6-1: Symboles EBNF

Définition 1 (*Diagramme SCT*) Un diagramme SCT est défini par les tuplets (S, T, V, c), tel que :

- ***S** est un état,*
- ***T** est une transition,*
- ***V** est une variable,*
- ***c** est une contrainte.*

Le langage SCT est en conséquence défini par la grammaire suivante :

<L_{CST}>::= "stateMachine" <ID> <S>+ <V>+ (<c>)? <T>+*

Tableau 6-2: Grammaire d'un graphe SCT

6.4.1. Syntaxe d'un état

Définition 2 (*État*) Un état S ∈ ∑S, où ∑S est l'ensemble de tous les états, est un concept abstrait pouvant être implémenté par le biais d'un état-contraintes simple (Simple State-Constraint : SSC en anglais) ou d'un état composite (Composite State : CS en anglais).

Définition 3 (*État composite*) Soit $\sum CS \subseteq \sum S$ un ensemble fini d'états composites et CS un élément de $\sum CS$. Un état composite peut être défini à plusieurs niveaux d'abstraction par un ensemble de sous-états. Les sous-états d'un état composite n (CS_n) constituent une région dénommée niveau de préoccupation (Concern Level : CL en anglais) dans cette thèse.

Définition 4 (*Niveau de préoccupation*) Un niveau de préoccupation j- aussi utilisé dans cette thèse sous sa dénomination anglaise *Concern Level* - Cl_{nj} est défini par sa projection dans $\sum S$, qui contient les sous-ensembles d'un flux d'exécution $\sum_{(k=0..i)} substate_{njk}$

Définition 5 (*État-Contraintes Simple*) Soit $\sum SSC$ un ensemble fini d'états-contraintes simples, et SSC un élément de $\sum SSC$. Un état-contraintes simple est défini par le tuplet SSC (s, V, c, ID) où :

- ***s** est le nom d'état*
- *V est un ensemble fini de variables contraintes par l'état (voir définition 7)*
- ***c** est une contrainte définie sur un ensemble de variables (voir définition 8)*
- ***ID** est un identifiant unique*

Les relations qui régissent les concepts S, SSC et CL sont définies par la grammaire suivante :

```
<S>::=<SSC>|<CS>
<CS>::="compositeState"<s><Cl>+"end_compositeState" [CS];
<Cl>::= "concernLevel" <ID> <S>+ "end_concernLevel" [Cl];
<SSC>::="simpleConstraintState" <s> <V>* <c>+
```

Tableau 6-3: Grammaire d'un graphe SCT - Les états

6.4.2.Syntaxe d'une transition

Définition 6 (*Transition*) $T \subseteq E \times S_S \times S_T$ est un ensemble fini de transitions, déplaçant le système d'un état source S_S à un état cible S_T. Il est déclenché par la survenue d'un événement E, défini par la grammaire du Tableau 6-4.

```
<Transition>::=("When"<E>)? ("Start"| "end")? "transition" ("from|to")? [S]
<E>::=<c>
```

Tableau 6-4: Grammaire d'un graphe SCT - Les transitions

6.4.3.Syntaxe d'une variable

Définition 7 (*Variable*) Soit V l'ensemble de toutes les variables contraintes par le graphe SCT, telles que V = $v_b \cup v_{enum} \cup v_{dom} \cup v_{att} \cup v_{card} \cup ctx$ où :

- $v_b \in \{0, 1\}$ *une variables booléenne*
- $v_{enum} \in \{enum\}$ *est une variables dont les valeurs* $\in \{enum\}$, *tel que* $\forall u \subseteq enum, u \in N \cup R$
- $v_{dom} \in dom\}$ *une variables dont les valeurs* $\in$ *dom, tel que* $\forall u \subseteq dom, u \in [minValue, maxValue]$
- $v_{att} \in N \cup R\}$ *une variables dont les valeurs sont entières ou des réelles*
- $v_{card} \subseteq \{0, 1\} \cup \{enum\} \cup dom \cup N \cup R$ est une variable multi-instanciée

Une variable est gouvernée par la grammaire introduite dans le Tableau 6-5.

<V>::= <v_b>|<v_{enum}>|<v_{dom}>|<v_{att}>|<v_{card}>
<v_b>::= "Boolean" "Param"? <ID>
<v_{enum}>::= ("Integer"|"Text") "Param"? "Enum" <ID> <Enumeration>
<v_{dom}>::= "Param"? <ID> <Domain>
<v_{att}>::= ("Integer"|"Real") "Param"? <ID>
<v_{card}>::="Param"? <ID> <multiple> <Domain>? <Enumeration>?
<multiple>::="[" <min> "," <max> "]"
<Domain>::= <RealDomain> | <IntDomain>
<Enumeration>::=<IntEnum>|<TextEnum
<RealDomain>::="[" <Real> ".." <Real> "]"
<IntDomain>::= "[" <Int> ".." <Int> "]"
<IntEnum>::= ("{<Int> ("," <Int>)+ "}")
<TextEnum>::= ("{"<Text> ("," <Text>)+"})

Tableau 6-5: Grammaire d'un graphe SCT - Les variables

6.4.4.Syntaxe d'une contrainte

Définition 8 (*Contrainte*) Une contrainte est une expression logique définie avec un ensemble de variables de V. Une contrainte peut être définie selon les règles énoncées dans le Tableau 6-6

<c>::=<LogicalExpression>

<LogicalExpression>::= <Statement> "⇒" | "⇔"| "⇐" | "V" | "∧" <Statement>

<Statement::=<Expression> ">="|"<="|"="|"!="|">"|"<"<Expression> | <Expression>

<Expression >::=<Atomic>"+|"<Expression>*

<Atomic>::= <C> | <C>""[V] | [V] "*" [V]*

Tableau 6-6: Grammaire d'un graphe SCT - Les contraintes

6.5. Contraintes syntaxiques

La syntaxe permet en effet de définir les concepts de modélisation dans un langage spécifique, et décrit le type de relations qui les unies les uns aux autres. Cependant, les bonnes pratiques de formalisation de langages recommandent la séparation des règles de syntaxe fondamentale des restrictions syntaxiques. Ces dernières font référence aux conditions syntaxiques qui doivent être satisfaites lors de la réalisation de ces modèles, et qui sont définies séparément, pour éviter d'encombrer la grammaire. Dès lors, les concepts des graphes états-contraintes transitions doivent remplir les conditions présentées dans le Tableau 6-7.

Restriction syntaxique	Commentaire
Soit $\sum CS^1$ l'ensemble des états composites de plus haute hiérarchie et $V=\{v_1, v_2, \dots v_n\}$ l'ensemble des variables du système. $\forall CS \in \sum CS^1$, $\forall v \in V$, v apparaît dans les contraintes réifiées par CS.	Un état composite racine contraint toutes les variables du système, car il correspond à un modèle de variabilité complet. L'état CdE par exemple, correspond au modèle de variabilité illustré dans la figure 6-3. Il représente un état global du SdS_Dy et de tous ses composants.
$\forall S \in \sum S$ et $\forall Cl \in \sum CL$, $\exists ! ID_S$ et $\exists ! ID_{Cl}$	Les identifiants des états et des niveaux de préoccupation sont uniques.
$\forall S \in \sum S$, tel que $\sum S$ est l'ensemble des états, $\exists T_1, T_2 \in \sum T$, avec $\sum T$ est l'ensemble des transitions, tel que S source de T_1, et S destination de T_2	Tout état est au moins un état source et un état destination d'une transition. C'est à dire que chaque état a un point d'entrée, qu'est la transition, qu'elle soit normale ou initiale. Idem, chaque état a un point de sortie, qui peut être une transition normale ou finale.

∀ CL∈∑CL, tel que ∑CL est l'ensemble des niveaux de préoccupations, $T_{Start} \in$ CL & $T_{End} \in$ CL.	Tout niveau de préoccupation possède une transition initiale (Start) et une transition finale (End). Cela permet de spécifier l'état qui s'active aussitôt que l'état composite parent est activé et l'état qui force la sortie de l'état composite parent lorsque celui-ci est activé.
∀ c ∈∑C, où ∑C est l'ensemble des contraintes, ∀ j ∈ {1, 2, .., n}, tel que v_j apparaît dans c, $v_j \in$ dom, avec dom ∈ Dom un domaine de déclaration d'une variable v_i.	Les contraintes s'appliquent en respectant le domaine de déclaration des variables. Une variable v déclarée dans un domaine [n,m], à titre d'exemple, ne devrait pas être soumise à une contrainte en dehors de ce domaine de définition, tel que v>m.

Tableau 6-7 : Restrictions syntaxiques

6.6. Sémantique

Le langage SCT décrit dans ce chapitre permet la spécification de lignes de produits logiciels dynamiques et évolutifs, à partir desquels plusieurs produits valides peuvent être dérivés. Par conséquent, le domaine sémantique du langage SCT, appelé S_{SCT}, est l'ensemble de tous les changements d'états valides, c'est à dire, l'ensemble des affectations valides des variables v dans leurs domaines respectifs. Autrement dit, le domaine sémantique est l'ensemble des produits valides.

Dans le langage SCT, la machine à états détermine la configuration du système. Elle représente une période de la vie du système pendant laquelle un ensemble d'exigences est satisfait. Comme ces exigences sont traduites en contraintes, l'exécution de la machine à états aboutit à la dérivation d'une solution valide (i .e., configuration) à partir d'un problème de satisfaction de contraintes. La machine à états finis admet l'orthogonalité (états pouvant être exécutés en parallèle), la hiérarchie (les super-états en tant que parents des sous-états) et la diffusion (événements sont diffusés dans l'ensemble du système) des états.

Un état est un concept abstrait qui spécifie, par le biais de contraintes, l'ensemble des exigences à satisfaire par le système concerné, dans un contexte spécifique. Les états peuvent être simples ou composites.

- **Les états-contraintes simples** sont des sous-états feuilles d'une machine à états. Ils n'englobent donc ni sous-états ni transitions. Chaque état simple correspond à un système individuel ou à un ensemble de systèmes. Il spécifie les exigences opérationnelles qui s'appliquent à ce système ou cet ensemble de systèmes, en faisant référence à des variables d'abstraction qui leur sont associées. Les exigences sont définies en termes de contraintes qui peuvent porter sur le système ou l'ensemble de systèmes. Les états-contraintes simples facilitent la spécification des exigences pour un nouveau système ajouté au dispositif spécifié en autorisant la manipulation de variables qui peuvent faire l'objet de contraintes qui portent de manière générique sur des systèmes connus à l'avance ou pas. Un état-contraintes simple est actif lorsqu'il est l'état cible d'une transition activée, et ce tant qu'aucune transition dont il est la source n'a été activée.
- **Les états composites** spécifient les exigences qui doivent être remplies lorsque la transition d'entrée de haut niveau est déclenchée. Ces exigences de haut niveau sont souvent décomposées en exigences de plus bas niveau et qui peuvent elles-mêmes être spécifiées par le biais d'autres états composites ou simples. Les états composites peuvent donc être considérés comme des super-états qui permettent de traiter plusieurs niveaux de préoccupation, auxquels on retrouve des états liés ou orthogonaux. Un état composite est actif lorsque :
  - Il est l'état cible d'une transition récemment activée ;
  - L'un de ses sous-états est l'état cible d'une transition récemment activée

Un niveau de préoccupation regroupe un ensemble de concepts (états, transitions, variables et contraintes) qui peuvent être analysés indépendamment des autres niveaux de préoccupation, ce qui aide à surmonter l'explosion combinatoire du nombre de transitions et l'augmentation non maîtrisée de la complexité des graphes décrivant des comportements complexes de systèmes. Différents niveaux de préoccupation formalisent des flux d'états composites pouvant être exécutés simultanément. En conséquence, les états appartenant à un même niveau de préoccupation sont des états séquentiels - c'est-à-dire qu'un seul peut être actif à la fois, alors que les états appartenant à différents niveaux de préoccupation peuvent être exécutés simultanément. Un niveau de préoccupation est activé lorsque:

- L'état composite parent est activé ;
- Un de ses états imbriqués est l'état cible d'une transition récemment activée.

Le graphe États-Contraintes Transitions (SCT)

Une transition guide la réponse du système à la manifestation d'un événement. Il s'agit d'une relation entre un état source et un état cible, qui est déclenchée lorsque l'événement décrit par la transition survient. Par conséquent, une transition est active lorsque l'événement est vrai. Deux cas spéciaux de transition existent :

- **Les transitions initiales** sont des transitions dépourvues d'événement déclencheur. Leur fonction principale est de désigner un point de départ pour un niveau de préoccupation, en se rapportant à l'état qui s'active automatiquement lorsque le niveau de préoccupation est activé.
- **Les transitions finales** sont des transitions qui mettent fin à l'exécution d'un niveau de préoccupation.

Un événement peut être détecté dans le SdS dyn ou dans son environnement, et est distribué dans l'ensemble de la machine à états qui le spécifient, tout événement étant potentiellement capable d'activer une transition. On distingue trois catégories d'événements : externes, internes, et temporels. Lorsqu'un événement est constaté dans l'environnement du système, on parle « d'événement externe ». Typiquement, les événements externes peuvent être traités par des processus externes. Leur détection opérationnelle peut être assurée par des senseurs, via des interfaces humain-machines, ou via des interfaces machine-machine,. Un événement peut également résulter d'un changement de la configuration (changement d'état) ou de la composition du SdS_Dy ; on parle alors « d'événement interne ». Enfin, les « événements temporels » sont déclenchés à des dates données qui peuvent être définies de différentes manière : dans l'absolu, en référence à d'autres événements, par régularité temporelle, etc.

Une variable est une abstraction qui porte sur un composant logiciel ou matériel, un sous-système, le système dans son entier, l'environnement d'un système, ou le temps. Les variables représentent les éléments qui affectent et/ou sont affectés par la configuration du système, en fonction des besoins exprimés ; elles peuvent avoir une valeur ou être indéfinies. Les exigences spécifiées dans chaque état sont traduites en contraintes appliquées à ces variables, sur des domaines finis.

Une contrainte est la représentation formelle d'une exigence spécifiée comme une expression régulière portant sur des variables contraignent leur valeur dans leurs domaines respectifs. Chaque état-contraintes simple spécifie au moins une contrainte. Ces contraintes doivent être satisfaites lorsque l'état auquel elles sont rattachées est actif. Lorsqu'un état n'est pas actif, les contraintes qui y sont rattachées peuvent être satisfaites.

6.7. Application

La Figure 6-2 présente le modèle de variabilité relatif à la solution GreenLife. Elle décrit les différents dispositifs d'irrigation, comme étant des éléments dont au moins un doit être sélectionné pour que la configuration du SdS_Dy soit valide. Les autres dispositifs qui contrôlent l'environnement sont considérés comme optionnels. Similairement, seuls les capteurs d'humidité sont obligatoires, les autres sont initialement optionnels. La source d'eau et le moyen de communication doivent impérativement faire partie de tous les produits dérivés du SdS_Dy d'irrigation.

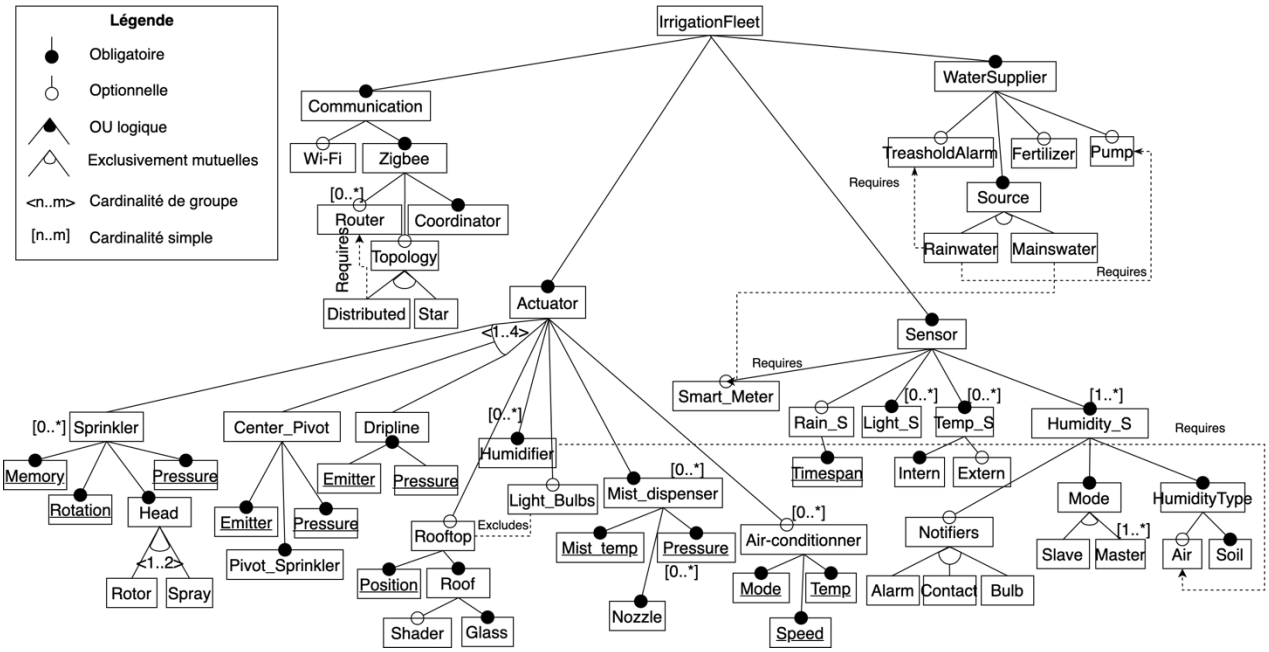

Figure 6-2: Modèle de variabilité de la solution GreenLife

6.7.1. Modèles de variabilité du SdS_Dy pour la cliente Maria

Maria, la propriétaire d'un produit dérivé de la solution GreenLife, souhaite faire usage de son système de manière spécifique, pendant les saisons d'automne et d'hiver, pendant les saisons du printemps et d'été et lorsqu'une coupure d'électricité se produit. Trois modes d'utilisation sont donc identifiés : Automne/Hiver (A/H), Printemps/Été (P/E), et Coupure d'électricité (CdE). Les exigences, relatives à chaque mode sont représentées par différents modèles de variabilité. Chacun de ces modèles décrit les dispositifs et dépendances impliqués dans le mode correspondant. Les Figure 6-3, 6-4 et 6-5 illustrent respectivement les modèles de variabilités considérés lorsque les modes CdE, A/H et P/E

sont actifs. Chaque modèle de variabilité représente, globalement, les contraintes que le SdS_Dy doit satisfaire, dans son mode respectif. De plus, la représentation graphique du langage SCT, accompagnant chacun de ces modèles, illustre la hiérarchie de chacun de ces modes, et présente un aperçu de la spécification formelle de ces exigences sous forme de contraintes exigées pour chaque état. Le modèle SCT, dans sa forme textuelle, relatif à la solution GreenLife, et spécifique aux exigences de Maria, est disponible en Annexe A.

La figure 6-3 illustre ce à quoi ressemblerait le modèle de variabilité du SdS_Dy de Maria, en mode NoE. Ce modèle est partiellement résolu (d'où les variables désactivées), mais comporte toujours de la variabilité, qui à son tour, et résolue dynamiquement par le modèle de reconfiguration (MrC). Le modèle MrC est ainsi accompagné d'une partie du modèle SCT, qui détermine les nouvelles contraintes à satisfaire pendant ce mode.

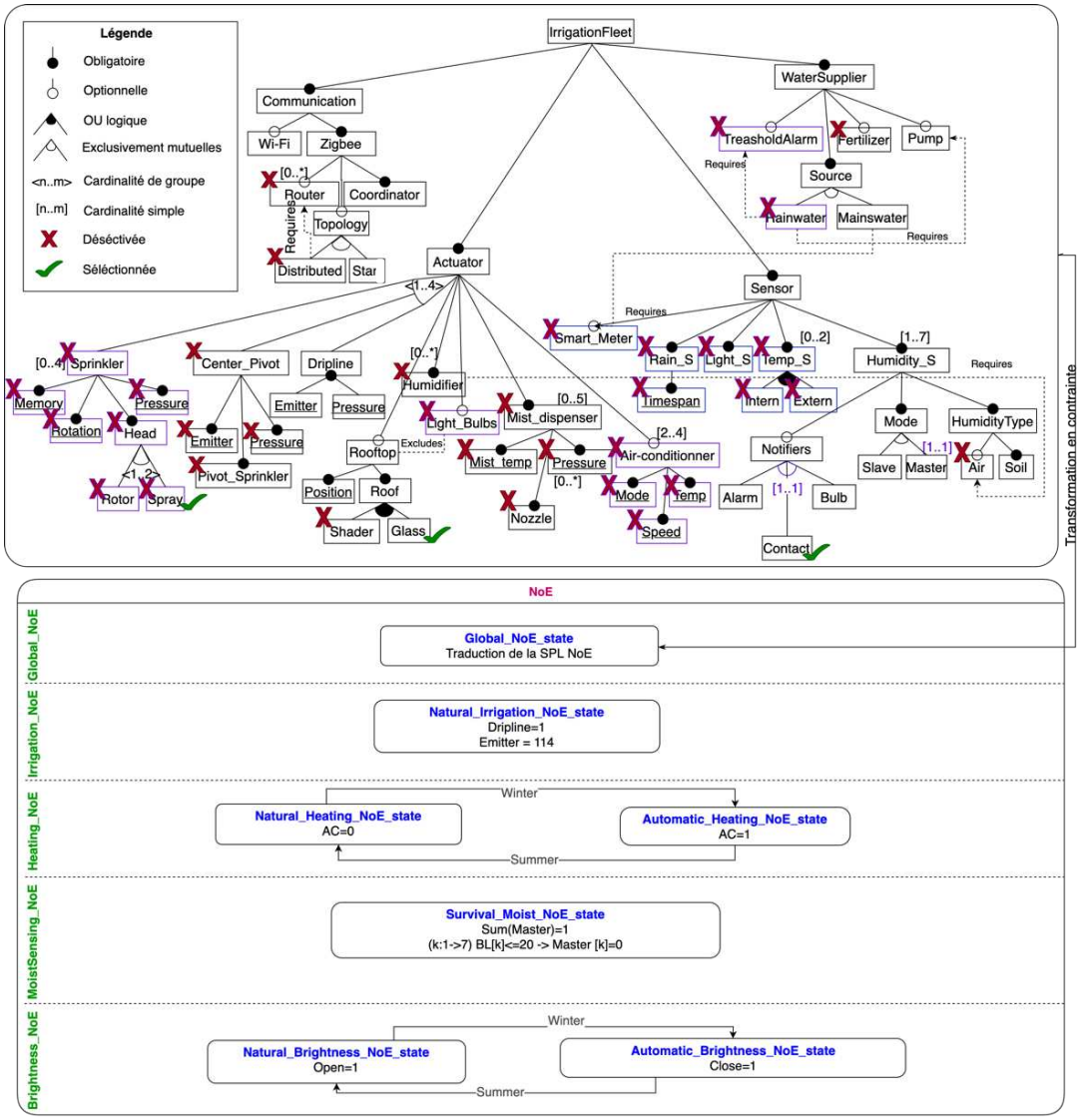

Figure 6-3: Représentation graphique des modèles MC et MrC pour le mode NoE

Similairement, La figure 6-4 illustre ce à quoi ressemblerait le modèle de variabilité du SdS_Dy de Maria, en mode SS, accompagné de la partie du modèle SCT qui spécifie les exigences dynamiques relatives à ce mode.

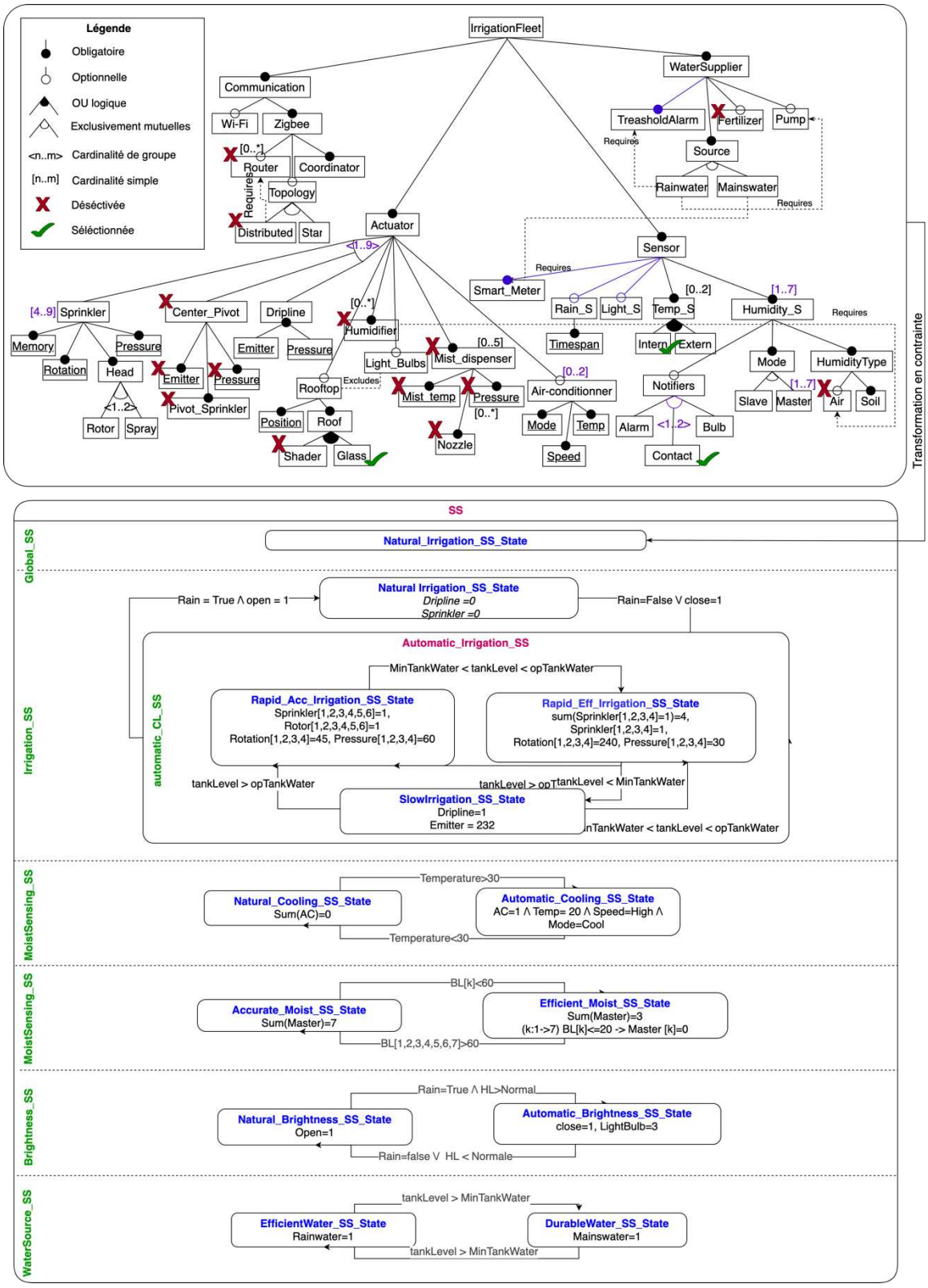

Figure 6-4: Modèle de variabilité, et graphe SCT pour le mode P/E (SS en Anglais)

Le graphe États-Contraintes Transitions (SCT)

Finalement, La figure 6-5 illustre ce à quoi ressemble le modèle de variabilité du SdS_Dy de Maria, en mode AW, accompagné de la partie du modèle SCT qui spécifie les exigences dynamiques relatives à ce mode.

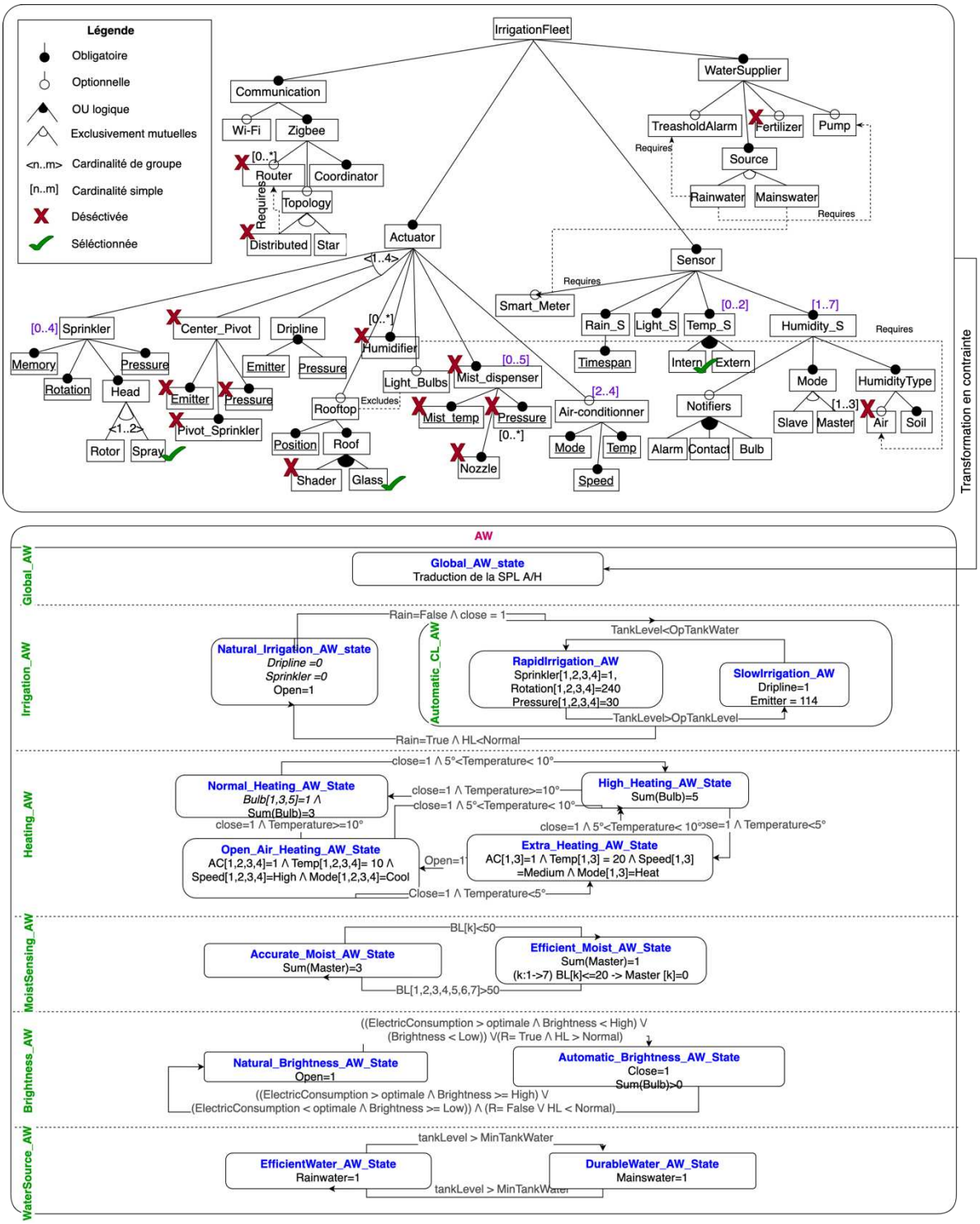

Figure 6-5: Modèle de variabilité, et graphe SCT pour le mode A/H (AW en Anglais)

Le modèle GreenLife selon la notation SCT est disponible en Annexe, et sur le lien suivant ci-dessous[4].

6.7.2.Spécification d'un système de systèmes dynamiques selon l'approche SCT

Après l'élaboration des exigences des utilisateurs d'un SdS_Dy, la spécification formelle de celles-ci, à travers le langage SCT, peut être effective, tel qu'il est décrit dans la Figure 6-6.

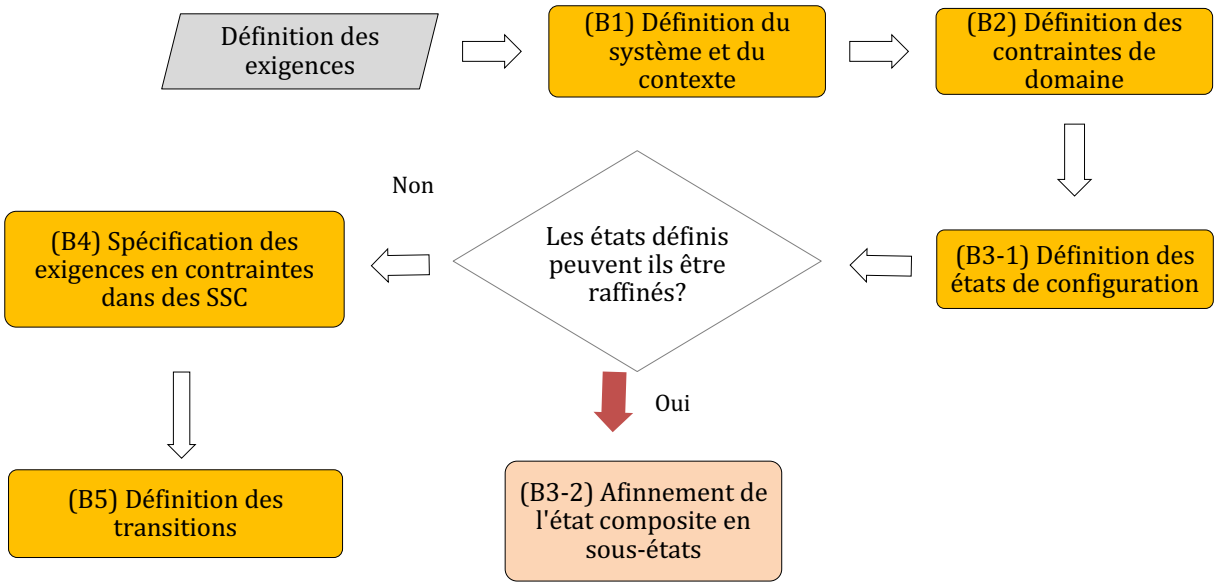

Figure 6-6: Processus de spécification via SCT

Le système ainsi que son contexte sont décrits sous forme de variables (B1), et la variabilité de domaine est spécifiée à travers des contraintes appliquées à celles-ci (B2). Les états de configuration qui spécifient la variabilité des exigences après l'exécution sont définis et affinés tant que possible en sous états de bas niveau (B3). Des contraintes sont élaborés dans les état-contrainte simple afin de spécifier formellement ces exigences (B4). Les conditions ou événements portant le SdS_Dy d'un de ces états vers un autre sont spécifiés à travers des transitions (B5).

[4] https://github.com/aachtaich/SCT-Models/blob/master/src/ThesisFleet.sct

Cette présentation séquentielle de la démarche est bien entendue indicative, et en déduire qu'il conviendrait d'appliquer la méthode de manière strictement séquentielle serait une erreur radicale. Comme dans toute présentation de démarche méthodologique, il convient de se concentrer sur la finalité des tâches à réaliser (le but), et lors de leur mise en œuvre, s'autoriser à mener les tâches en parallèle, itérer, incrémenter, etc. lorsque cela s'avère nécessaire, et ce jusqu'à ce que les buts soient atteints. La section suivante décrit les taches impliquées dans ce processus de modélisation.

6.7.2.1. Définition du système et du contexte (B1)

Le modèle SCT peut être utilisé exclusivement ou peut accompagner un modèle LdP externe. Dans le premier cas, les éléments du système sont déclarés en tant que variables, suivant la forme définie par la grammaire présentée plus haut. Dans le second cas, ces éléments sont importés à partir du modèle externe, en adoptant la grammaire du langage SCT. Les Figure 6-7 et Figure 6-8 introduisent la déclaration des éléments du système et du contexte, respectivement.

```
%Declaration of variables
Boolean Fleet
Boolean Actuator
Boolean Sprinkler [1,9]
Boolean Memory [1,9]
Integer Enum Rotation [1,9]{0, 45, 90, 180, 360}
Boolean Head [1,9]
Integer PressureSp [1,9] [0..60]
Boolean Rotor [1,9]
Boolean Spray [1,9]
Boolean CenterPivor
Integer Enum EmitterCP {0, 114,232}
Integer PressureCP [0..60]
Boolean SprinklerCP
Boolean Dripline
Integer Enum Emitter {0, 114,232}
Integer PressureDr [0..60]
Boolean Humidifier
Boolean Rooftop
Boolean Position
Boolean OUT_open
Boolean OUT_close
Boolean Partial
Boolean Roof
Boolean Shader
Boolean Glass
Boolean LightBulb [1,5]
Boolean MistDispenser
Boolean MistTemp
Boolean PressureMD
Boolean Nozzle
Boolean AirConditionner [1,4]
```

Figure 6-7 : Déclaration des variables des éléments selon le langage SCT (SCT_MV)

```
1  %Context variables
2  Float Param BatteryLevel[1,7]
3  Integer Param Temperature
4  Float Param Brightness
5  Float Param minBrightness
6  Float Param opBrightness
7  Float Param TankLevel
8  Float Param MinTankWater
9  Float Param OpTankWater
10 Float Param WaterConsumption
11 Boolean Param Rain
12 Float Param EConsumption
13 Float Param OpConsumption
14 Boolean Param PowerFailure
15 Integer Param Month
16 Float Param HumidityLevel
17 Float Param NormalHumidity
```

Figure 6-8: Déclaration du contexte selon le langage SCT (SCT_MDC_&_MCA)

La première figure définit la LdP relative à la solution GreenLife, dont les éléments sont spécifiés comme variables dans leurs domaines respectifs (e.g. Integer Enum Rotation [1,9]{0,45,90 ,180 ,360} —Ligne6—, Boolean Dripline —Line 15—, Integer PressureDr [0..60] —Ligne 17—). La deuxième figure définit les éléments de contexte pertinents pour Maria, comme le niveau de batterie des capteurs (e.g. Float Param BatteryLevel[1,7] —Ligne 2—) ou le mois de l'année (e.g. Integer Param Month —Ligne 15—), ainsi que les variables qui doivent être paramétrées au fur et à mesure que le SdS_Dy est exécuté (e.g. Float Param OpBrightness —Ligne 6—)

6.7.2.2. Définition des contraintes de domaine (B2)

Suite à la définition du système sous forme de variables, des contraintes sont appliquées à celles-ci afin de spécifier les règles de variabilité et dépendances qui les régissent. Dans un modèle SCT, ces contraintes sont éventuellement transformées en contraintes flexibles, afin de ne pas être en contradiction avec les nouvelles contraintes introduites, qui sont plus pertinentes pour l'utilisateur du produit dérivé. La Figure 6-9 présente l'exemple de quelques contraintes définies pour le SdS_Dy GreenLife.

```
1 %External constraints
2 Fleet=Communication
3 Fleet=Actuator
4 Fleet=Sensor
5 Actuator≥Humidifier
6 Actuator≥LightBulb[1] /\ Actuator≥LightBulb[2] /\ ...
      Actuator≥LightBulb[3] /\ Actuator≥LightBulb[4] /\ ...
      Actuator≥LightBulb[5] /\ Actuator=1 -> ((LightBulb[1]+ ...
      LightBulb[2]+ LightBulb[3]+ LightBulb[4]+ LightBulb[5])>0 ...
      /\ (LightBulb[1]+ LightBulb[2]+ LightBulb[3]+ ...
      LightBulb[4]+ LightBulb[5])<5)
7 Humidifier>0 -> ...
      (Air[1]+Air[2]+Air[3]+Air[4]+Air[5]+Air[6]+Air[7]>0)
8 OUT_open * (LightBulb[1]+ LightBulb[2]+ LightBulb[3]+ ...
      LightBulb[4]+ LightBulb[5]) =0
```

Figure 6-9 : Contraintes du domaine selon le langage SCT (SCT_MV)

Les Lignes 2, 3 et 4 par exemple sont des contraintes qui spécifient que pour tout système dérivé, un moyen de communication doit être spécifié, ainsi que des capteurs et des actuateurs. La ligne 5 définit que l'actuateur Humidifier est optionnel. La contrainte de la ligne 6 spécifie l'exigence qui stipule que dans tout produit dérivé de la solution GreenLife, un maximum de 5 ampoules peuvent être actifs à la fois. Finalement, la ligne 8 décrit la relation d'exclusion mutuelle entre une toiture ouverte, et les ampoules.

6.7.2.3. Définition des états de configuration (B3)

En accord avec les exigences dynamiques des utilisateurs, un système, un périphérique et même un composant logiciel peuvent être dans différents états de configuration. Par conséquent, à ce stade de la modélisation selon l'approche SCT, une hiérarchie des états est définie, en partant des états de haut niveau jusqu'aux états pouvant être opérationnalisés. La Figure 6-10 présente un échantillon de la hiérarchie des états définit pour la solution de Maria.

En effet, les états qui se produisent en parallèle, qui sont généralement liés à un sous-ensemble spécifique de périphériques ou dont l'occurrence est liée à des événements corrélés sont regroupés par niveau de préoccupation. Par exemple, dans l'état composite AW —Ligne 2, les états qui dépendent de l'humidité du sol (Lignes 8, 10, 12 et 14) et de la température (Ligne 20) sont regroupés dans deux niveaux de préoccupations différents et orthogonaux, à savoir concernLevel Irrigation_AW —Ligne 7— et concernLevel Heating_AW —Ligne 19—.

```
1  %Statemachine hierachy
2  compositeState AW
3      concernLevel global-AW
4          simpleConstraintState global-AW-state
5              {Constraint}
6      end-concernLevel global-AW
7      concernLevel irrigation-AW
8          simpleConstraintState Natural-Irrigation-AW-state
9              {Constraint}
10         compositeState automatic-AW
11             concernLevel automatic-CL-AW
12                 simpleConstraintState rapidIrrigation-AW
13                     {Constraints}
14                 simpleConstraintState slowIrrigation-AW
15                     {Constraints}
16             end-concernLevel automatic-CL-AW
17         end-compositeState automatic-AW
18     end-concernLevel irrigation-AW
19     concernLevel heating-AW
20         simpleConstraintState normal-heating-AW-state
21             {Constraint}
22         ...
23 end-compositeState AW
24 compositeState NoE
25     concernLevel global-NoE
26         simpleConstraintState global-NoE-state
27             {Constraint}
28     end-concernLevel global-NoE
29     concernLevel irrigation-NoE
30         simpleConstraintState Natural-Irrigation-NoE-state
31             {Constraint}
32     end-concernLevel irrigation-NoE
33     ...
34 end-compositeState NoE
35 ...
```

Figure 6-10: Échantillon de la hiérarchie du modèle SCT (SCT_MrC & SCT_MC)

Ensuite, les différents états de configuration du SdS_Dy d'irrigation, pour les modes Automne/Hiver (A/H), en anglais (AW) et Coupure d'Électricité (CdE), en anglais NoE. Ceux-ci sont représentés par les états composites `compositeState AW` —Ligne 2— et `compositeState NoE` —Ligne 24, constitués à leur tour de sous états-contraintes simples et des états composites tels que `simpleConstraintState natural-heating-AW-state` —Ligne 8— et `compositeState automatic.AW` —Ligne 10—.

6.7.2.4. Conversion des exigences en contraintes (B5)

Un état correspond à la spécification des exigences qui doivent s'appliquer au système dans des circonstances spécifiques. Le système est décrit abstraitement via un ensemble de variables, et les exigences imposées pour les différents états du système sont spécifiées sous forme de contraintes sur ces variables. L'activité de conversion des exigences en

contraintes consiste à spécifier les exigences dans chaque état-contrainte simple, en tant que contraintes tel qu'illustré dans la figure suivante.

```
1 %Constraints withinh simple-Constraint States
2 simpleConstraintState automatic_cooling_SS_state
3 AirConditionner[1]+AirConditionner[2]+ AirConditionner[3]+ ...
      AirConditionner[4]=1
4 Temp[1]+Temp[2]+Temp[3]+Temp[4]=20 /\ Temp[1]*Temp[2]=0 /\ ...
      Temp[3]*Temp[4]=0 /\ Temp[1]*Temp[3]=0 /\ Temp[2]*Temp[4]=0
5 Speed[1]+Speed[2]+Speed[3]+Speed[4]=3 /\ Speed[1]*Speed[2]=0 ...
      /\ Speed[3]*Speed[4]=0 /\ Speed[1]*Speed[3]=0 /\ ...
      Speed[2]*Speed[4]=0
6 Mode[1]+Mode[2]+Mode[3]+Mode[4]=2 /\ Mode[1]*Mode[2]=0 /\ ...
      Mode[3]*Mode[4]=0 /\ Mode[1]*Mode[3]=0 /\ Mode[2]*Mode[4]=0
```

Figure 6-11: Exemple de contraintes dans les SimpleConstraintState selon le langage SCT (SCT_MrC)

La Figure 6-11 présente à partir de la ligne 3 les exigences de refroidissement automatique, mises en œuvre lorsque l'état SS est actif, suite à la mesure d'une température supérieure à 30°. Ainsi, lorsque l'état automatic_cooling_SS_state est activé, parmi les quatre climatiseurs présents dans le SdS_Dy de la cliente Maria, un seul doit être activé, avec une température égale à 20°, une vitesse rapide, et en mode refroidissement.

6.7.2.5. Définition des transitions (B4)

Les transitions définissent les règles de passage d'un état source à un état destination. Elles permettent de spécifier la dynamique des exigences en définissant les événements et/ou conditions qui convertissent des exigences requises à un moment (t) par un nouvel ensemble d'exigences requises au moment (t+1). La Figure 6-12 présente un échantillon de transition.

```
1 %Transitions
2 Start transition to NoE.heating_NoE.natural_heating_NoE_state
3 Start transition to NoE.moitSensing_NoE.survival_moist_NoE_state
4 When NoE if (Month=10) transition from ...
      NoE.heating_NoE.natural_heating_NoE_state to ...
      NoE.heating_NoE.automatic_heating_NoE_state
5 When NoE if (Month=4) transition from ...
      NoE.heating_NoE.automatic_heating_NoE_state to ...
      NoE.heating_NoE.natural_heating_NoE_state
```

Figure 6-12 : Exemple de transitions selon le langage SCT

La figure décrit d'abord les états qui sont automatiquement activés lorsque l'état composite NoE est actif (lignes 2 et 3). Elle présente ensuite la syntaxe concrète des transitions entre le chauffage automatique et le chauffage naturel pour le mode NoE, en précisant les événements déclencheurs.

6.8. Conclusion

Le langage SCT formalisé dans ce chapitre est un langage de spécification d'exigences de systèmes de systèmes dynamiques, appliqué au cas de la solution GreenLife, un système d'irrigation. Basé sur le formalisme des automates à états finis, le langage SCT élargit ces modèles par la notion de contraintes, notamment en spécifiant les exigences imposées sous forme de contraintes. Cela étend le pouvoir d'expression, en combinant la facilité d'utilisation d'un langage établi, c'est à dire les graphes états transitions, avec la capacité d'analyse de la programmation par contraintes. Grâce à ce nouveau formalisme, les exigences dynamiques du SdS_Dy d'irrigation propre à Maria ont été spécifiées avec succès. Cette modélisation commence par définir les éléments du système et de son contexte, à spécifier les exigences de domaine sous forme de contraintes flexibles, à définir la hiérarchie du modèle SCT en termes d'états simples et composites et de niveaux de préoccupation, et finalement, à identifier les exigences qui doivent être satisfaites au niveau de chaque état-contraintes simple, et les traduire vers leur représentation en contraintes.

Chapitre 7 Configurations dynamiques

7.1. Introduction

La configuration, dans le domaine de l'ingénierie des lignes de produits, fait référence à la dérivation d'un produit valide. La configuration d'un SdS_Dy, spécifié comme LdP, représente un produit dérivé valide, conformément aux exigences et au contexte et qui comporte tous les éléments du SdS_Dy sélectionnés pour cette configuration. Le processus de configuration se base ainsi sur l'analyse des exigences des utilisateurs, pour dériver un produit comportant les éléments obligatoires, communs à toutes les configurations, ainsi que certains éléments variables, réalisant les exigences retenues.

Parmi les approches de configuration automatique existantes (Benavides et al. 2010), la programmation par contraintes est une démarche basée sur une logique formelle mathématique pour analyser les lignes de produits et assister le processus de configuration (Batory 2005; Czarnecki and Kim 2005; Mannion et al. 2009; Classen, Boucher, and Heymans 2011; Sawyer et al. 2012a; Benavides et al. 2005; Sawyer et al. 2012b; Achtaich et al. 2019). Elle représente dans ce chapitre, une ébauche à la formalisation de la sémantique du langage SCT, à travers la transformation de ses concepts, et en conséquence, du modèle SCT, en programme de contraintes, dont la résolution constitue les éléments d'une configuration conforme aux exigences, transformées en contraintes.

Le chapitre en cours introduit les règles de transformations prises en compte pour traduire un modèle SCT en problème de contraintes. Il décrit ensuite la structure du langage ainsi que l'outil Xtext, employé pour son implémentation. L'exemple d'exploration GreenLife, pour la cliente Maria est exploité pour illustrer le programme de contraintes généré du modèle SCT relatif à cette solution, ainsi que les résultats de la résolution de ce problème de contraintes pour des exigences et contextes variant.

7.2. Programmation par contraintes

La programmation par contraintes (Rossi et al. 2006) est une approche logique déclarative, utilisée pour résoudre des problèmes du monde réel. Elle est utilisée comme formalisme de spécification de systèmes et de processus, décrits sous forme de variables. Contrairement aux approches procédurales, qui définissent un algorithme décrivant les instructions nécessaires à la résolution d'un problème spécifique, un programme de

contraintes définit les propriétés nécessaires à la solution et délègue les tâches de prise de décision à un solveur.

Selon cette approche, le programme s'appelle un problème de satisfaction de contraintes (PdSC). Il définit le problème en termes de variables et de dépendances logiques qui renseignent les valeurs valides de ces variables dans leurs domaines respectifs. L'objectif d'un programme de contraintes est d'affecter une valeur à une variable, de manière à satisfaire toutes les contraintes définies. Ceci est réalisable d'une part, grâce au principe de propagation, qui consiste à éliminer au fur et à mesure, les domaines et les valeurs conflictuelles, et d'autre part, au moyen d'affectation de valeurs, en respectant des heuristiques prédéfinies (la première valeur possible, la valeur réalisant un optimum, etc.)

Les problèmes de satisfaction des contraintes statiques (*hard constraints* en anglais) peuvent s'avérer efficaces pour la spécification de lignes de produits statiques. Sachant que les SdS_Dy subissent des modifications constantes, même après leur spécification initiale, raisonner sur de tels environnements dynamiques va au-delà d'un programme de contraintes statiques, car ils stipulent que l'ensemble des variables et des contraintes est connu et fixé à l'avance. Les problèmes de satisfaction de contraintes dynamiques (*Dynamic Constraint Satisfaction Problem* : DCSP en anglais) fournissent les mécanismes nécessaires pour analyser progressivement différents ensembles de variables et de contraintes pour le même problème (Dechter and Pearl 1988). De plus, dans la plupart des cas, les problèmes de satisfaction de contraintes typiques sont composés de contraintes rigides, qui doivent obligatoirement être satisfaites. Cependant, les problèmes du monde réel sont de nature plus modulable. Le concept de contraintes flexibles a été introduit pour permettre leur spécification, et pour éviter des problèmes excessivement contraints. En s'appuyant sur cela, des solutions valides peuvent être générées d'un PdSC, même lorsque certaines des contraintes flexibles ne peuvent être satisfaites (Freuder and Wallace 1992).

Prenant en considération ces deux aspects, le langage SCT doit prendre en charge les contraintes flexibles et dynamiques, afin d'offrir aux systèmes spécifiés la possibilité d'évoluer dans le temps et dans l'espace.

7.3. Scenarios de reconfiguration

La reconfiguration est la capacité d'un SdS_Dy à activer, désactiver et mettre à jour des services à la demande ou en réponse à des fluctuations internes ou externes. Les notations de spécification de tels systèmes se concentrent essentiellement sur les changements environnementaux qui impactent les configurations. Les exigences et compositions dynamiques elles, sont plus complexes à gérer, car elles sont plus difficiles à surveiller et peuvent mener à des résultats erronés ou encore provoquer des contradictions.
Plusieurs scénarios de reconfigurations sont pris en compte dans la notation SCT. Des reconfigurations qui proviennent suite à un changement dans l'environnement du SdS_Dy (a et b), des reconfigurations qui se produisent à l'issue de la modification dans sa composition (c et d) ou des reconfigurations qui surviennent en conséquence d'une révision des exigences (e et f).

a) **Lorsqu'un événement contextuel se produit:** Le contexte peut être interne ou externe (e.g. Perte de connectivité, changement de température), mais aussi temporels (e.g, mois, heure, date limite). Les événements contextuels sont décrits dans la notation comme des déclencheurs de transition, et conduisent le SdS_Dy d'un état à un autre, et donc d'un ensemble d'exigences qui doivent être satisfaites à un autre. Une nouvelle configuration ne se produit que lorsque les contraintes qui traduisent le nouvel ensemble d'exigences contredisent les contraintes précédentes.
b) **Lorsqu'une variable paramétrique est modifiée:** Les variables paramétriques sont des abstractions d'une partie du SdS_Dy qui peuvent être mises à jour au moment de l'exécution (e.g, les valeurs optimales pour l'irrigation sont mises à jour par Maria, lorsqu'elle décide de convertir son champs en plantation de pomme de terre). Parallèlement à la ré-instanciation de ces variables, et en raison des dépendances qu'elles possèdent avec d'autres variables du système ou du contexte, un recalcule de contraintes est effectué, et une reconfiguration se produit en conséquence. Ce calcul s'appuie sur le principe de propagation de contraintes pour déterminer un nouveau domaine de valeurs valide qui satisfasse l'ensemble des contraintes, à la lumière des modifications administrées.
c) **Lorsqu'un nouveau périphérique est introduit:** un nouvel appareil est spécifié en définissant un ensemble de nouvelles variables qui lui font référence ainsi qu'à ses éléments imbriqués, et ses différents modes d'emploi (e.g, Maria introduit un nouvel

appareil d'irrigation; un centre-pivot. Celui-ci possède une tige d'arrosage qui peut avoir un nombre de sorties variant, et qui pompe l'eau avec la pression souhaitée). Bien que les exigences demeurent inchangées (e.g. Le système d'irrigation doit utiliser un et un seul dispositif d'irrigation pour maintenir l'humidité du sol), les composants peuvent être soumis à une reconfiguration afin de demeurer conforme à cette exigence (e.g. Le centre Pivot est maintenant actif au lieu des Sprinkler, du fait qu'il assure —avec le meilleur score— la durabilité de service).

d) **Lorsqu'un périphérique est supprimé**: Des composants, des systèmes ou des dispositifs peuvent être supprimés du SdS_Dy au cours de son exécution, car ils risquent de devenir inutiles, défectueux, compromettant aux performances requises ou tout simplement obsolètes (e.g. En attendant qu'il soit réparé, Maria décide de supprimer un capteur d'humidité de l'air défectueux). De façon similaire au scénario de reconfiguration précédent, alors que les exigences demeurent stables, des reconfigurations peuvent toujours se produire (e.g. En raison de sa dépendance au capteur d'humidité de l'air, l'humidificateur est mis en veuille prolongée).
e) **Lorsque des exigences sont introduites**: Des exigences nouvelles peuvent être ajoutées aux états existants (e.g. Il est observé que Maria démarre manuellement l'arrosage par sprinkler plus souvent que la configuration indiquée, des exigences qui se rapportent à la priorité des arroseurs sont ainsi définies dans les états propres à cet effet), ou définies dans un état composite nouveau (e.g. nouveau propriétaire du champ possède une vision différente de celle de Maria). Ce nouvel ensemble d'exigences se traduit par des nouvelles contraintes. Leur satisfaction implique manifestement la reconfiguration du SdS_Dy.
f) **Lorsqu'un ensemble d'exigences existant est supprimé**: La suppression des exigences réduit le nombre de contraintes appliquées aux variables du système. Cela élargit l'espace de solution en admettant de nouvelles valeurs et, par conséquent, de nouvelles configurations possibles.

7.4. Règles de transformation d'un modèle selon la notation SCT

Dans le but d'atteindre une configuration qui satisfasse les exigences des utilisateurs, le modèle, selon la notation SCT est transformé en programme de contraintes. L'exécution de ce dernier est la première étape du processus de configuration. Les neufs règles

présentées ci-dessous sont utilisées pour transformer les concepts du modèle en contraintes.

- **Règle1:** tous les identifiants sont représentés en tant que variables booléennes.

Règle 1	Soient ∑S l'ensemble des états et ∑CL l'ensemble des niveaux de préoccupation : $\forall e \in \sum S \times \sum CL,\ e \in \{0,1\} \times \{0,1\}$
Exemple	**var 0..1 : automatic_CL_AW** (Niveau de préoccupation) **var 0..1 : rapidIrrigation_AW** (État-contrainte Simple) **var 0..1 : automatic_AW** (État composite)

- **Règle 2 :** toutes les variables sont transformées en variables déclarées dans leurs domaines respectifs.
- **Règle 3 :** un niveau de préoccupation est activé si et seulement si l'état composite parent est activé.

Règle 3	Soient CS_n un état composite n et $\sum_{(k=0..i)} Cl_{nk}$ les i niveaux de préoccupation appartenant à l'état composite CS_n $CS_n = 1 \Leftrightarrow cl_{n1} = cl_{n2} = \ldots = cl_{ni} = 1$
Exemple	`constraint AW=1 ⇔ moitSensing-AW=1;` `constraint AW=1 ⇔ brightneAW-AW=1;`

- **Règle 4 :** les états appartenant au même niveau de préoccupation peuvent être activés de manière séquentielle.

Règle 4	Soient Cl_{nx} un niveau de préoccupation, et $\sum_{(k=0..j)}$ $state_{nxk}$ l'ensemble des états constituants Cl_{nx} $Cl_{nx} = \sum_{(k=0..j)} state_{nxk}$
Exemple	`constraint` brightneAW-AW = natural-brightness-AW+ automatic-brightness-AW

- **Règle 5 :** les états appartenant à différents niveaux de préoccupations dans un état composite peuvent être exécutés simultanément.

Règle 5	Soient CS_n un état composite n, $\sum_{(k=0..i)} Cl_{nk}$ l'ensemble des niveaux de préoccupation appartenant à l'état composite CS_n et $\sum_{(k=1..i)} \sum_{(l=1..j)} state_{nkl}$ l'ensemble des états imbriqués dans l'état composite CS_n :

	$\sum_{(k=1..i)} \sum_{(l=1..j)} state_{nkl} = CS_n * \sum_{(k=0..i)} Cl_{nk}$
Exemple	`constraint automatic-CL-SS = automatic-SS * ( automatic-CL-SS );` `constraint global_NoE + irrigation_NoE + heating_NoE + moitSensing_NoE = NoE * ( global_NoE + irrigation_NoE + heating_NoE + moitSensing_NoE );`

- **Règle 6 :** un état est actif s'il s'agit de l'état cible d'une transition récemment activée

Règle 6	Soient T_{nm} une transition porteuse de l'événement E, et S_{no} son état cible : $E \Rightarrow S_{no} = 1$
Exemple	`constraint (((MinTankWater<TankLevel ∧ TankLevel> OpTankWater)) ∧ (SS=1 )) ⇒ rapidEff-Irrigation-SS = 1;)`

- **Règle 7 :** lorsqu'un état composite est actif, l'état de début est activé, sauf si la règle 3 est valide.

Règle 7	Soient CS_n un état composite, Cl_{nx} un niveau de préoccupation, $(\sum_{(k=0..i)} state_{nxk}$ l'ensemble de ses états, S_{nxy} l'état initial : $For_{(k=1..j)}$, where $\lvert Cl_n \rvert = j$, { $(CS_n = 1 \Rightarrow S_{nxy} = 1)$ & $(\sum_{(k=0..i)} state_{nxk} = 1)$ }
Exemple	`constraint SS = 1 ⇒(efficientWater-SS-state =1 ∨ efficientWater-SS-state + durableWater-SS-state= 1);`

- **Règle 8 :** toutes les contraintes extraites de modèles externes sont transformées en contraintes flexibles

Règle 8	Soient C_{ext} une contrainte externe et $Flex_k$ une variable booleanne : $\forall\ c_{ext}$, $For_{(k=1..j)}$, where $\lvert c_{ext} \rvert = j$, { $Flex_k \Leftrightarrow c_{extk}$ }
Exemple	`constraint Flex21 = 1 ⇔ (MistDispenser>0 ⇒ PressureMD >0);`

- **Règle 9 :** lorsqu'un état est actif, toutes ses contraintes intégrées doivent être vérifiées

Règle 9	Soient SSC_{nxy} un état, $\sum_{(k=1..p)} c_{nxyp}$ ses contraintes : For (k=1..p) $S_{nxy} \Rightarrow c_1 \wedge c_2 \wedge .. \wedge c_{nxyp}$
Exemple	`constraint automatic-cooling-SS-state = 1 ⇒ ( (AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4]=1) ∧`

	(Temp[1] + Temp[2]+ Temp[3]+ Temp[4]=20 ∧ Temp[1]*Temp[2]=0 ∧ Temp[3]*Temp[4]=0 ∧ Temp[1]*Temp[3]=0 ∧ Temp[2]* Temp[4]=0;)

7.5. Implémentation

Le langage SCT est défini sur deux volets. Sa grammaire introduit les concepts qui le caractérisent et les règles de transformations présentées ci-dessus définissent leur rapport au domaine sémantique, qui dans cette thèse est la programmation par contraintes. Cette section présente l'implémentation préliminaire du langage à travers le Framework Xtext. Cette implémentation a pour but de faciliter la spécification de SdS_Dy en permettant la création de modèles et en générant en sortie le programme de contraintes correspondant afin de soutenir les opérations de reconfiguration.

7.5.1. Outillage

Les outils pour la création de langages dédiés (*Domain Specific Langages* : DSL en anglais) permettent d'augmenter le niveau d'abstraction de nouveaux langages. Xtext est un exemple de Framework qui facilite la création de langages dédiés, en offrant le support nécessaire pour la définition et la validation d'une grammaire, ainsi que la génération de code exécutable. Particulièrement, Xtext est reconnu pour sa simplicité d'utilisation, mais aussi, sa complétude (Bettini 2013), car il ne s'agit pas simplement d'un analyseur syntaxique, mais d'un environnement de développement complet. Le langage SCT est donc implémenté à l'aide de ce Framework, comme illustré dans la Figure 7-1.

- **Méta-niveau formel :** la première étape dans la définition du langage est la définition formelle de sa syntaxe et sa sémantique. La première peut être définie avec une grammaire ou à l'aide de méta-modèle, et la deuxième peut être définie à travers un langage informel ou à l'aide de logique mathématique. Dans cette thèse, une grammaire ainsi qu'une définition textuelle en langage naturel de chaque concept sont utilisées pour présenter la syntaxe et la sémantique de SCT, respectivement.

Méta-Niveau : la syntaxe est implantée avec Xtext. Celui-ci permet la définition des règles de grammaire, afin de créer des modèles corrects et cohérents avec la définition formelle.

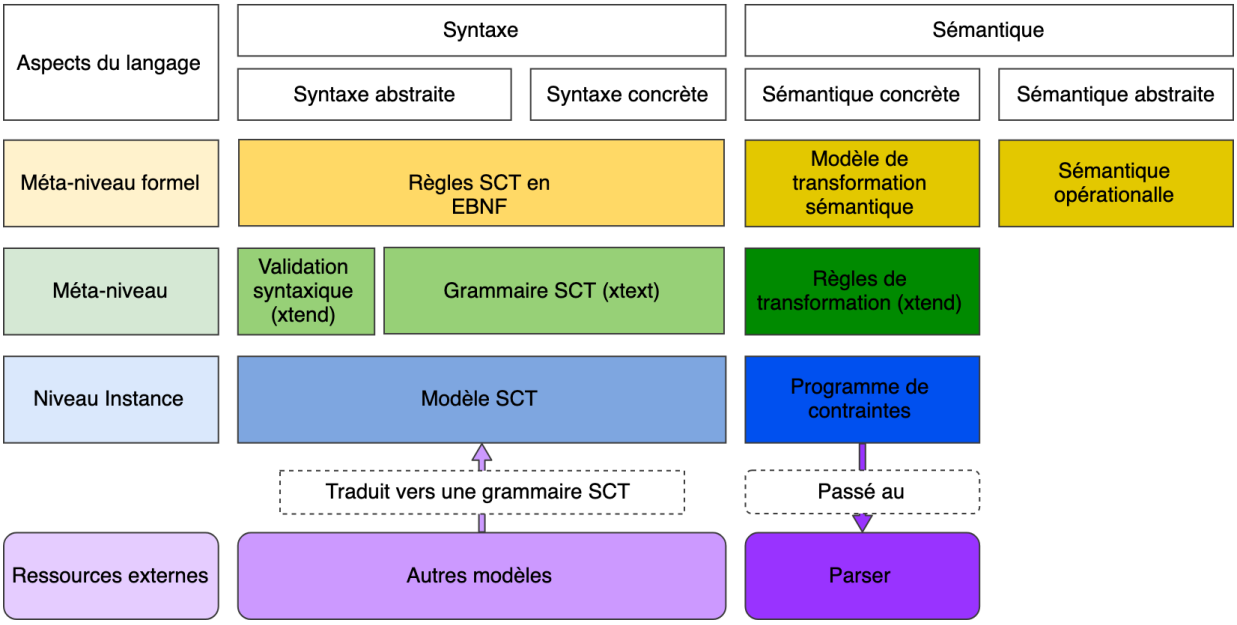

Figure 7-1: Outillage de SCT avec Xtext

Pour éviter de surcharger cette grammaire, un validateur indépendant est mis en œuvre en parallèle avec le langage Xtend, pour définir les contraintes syntaxiques. Ensuite, les règles de transformation, introduites dans la section 7.3, sont exprimées à l'aide du module Xtend, dans le but d'automatiser la génération de code.

- **Niveau instance :** l'exécution de la grammaire permet la création de modèles SCT, qui, une fois compilés, génèrent un programme de contraintes en format (.mzn) dans un éditeur Minizinc (Minizinc 2019). Le choix de cet éditeur est basé principalement sur la facilité de son utilisation et son indépendance du solveur de haut niveau.
- **Ressources externes :** D'un côté, des modèles externes, traduits conformément à la syntaxe SCT, peuvent être importés dans celui-ci. D'un autre côté, le modèle de contraintes généré est exécuté grâce à un solveur externe, qui s'occupe de trouver une solution valide, en considérant toutes les contraintes.

Le méta-niveau correspondant à l'implémentation de la grammaire et des règles de transformation du langage SCT sont disponibles sur le lien suivant : https://github.com/aachtaich/SCT

7.5.2. Transformation d'un modèle selon la notation SCT en CSP

Suivant les règles de transformations définies préalablement, un modèle selon la notation SCT est traduit en contraintes, constituant un problème de contraintes. Le programme de contraintes gé néré du modèle SCT propre au système d'irrigation GreenLife est

disponible sur le lien suivant à titre illustratif : https://github.com/aachtaich/SCT-Models/blob/master/src-gen/ThesisExample.mzn.

Pour commencer, les variables décrivant le SdS_Dy sont définies dans leur domaines respectifs. Les Figure 7-2 et Figure 7-3 illustrent la représentation des variables de système et de contexte, introduites préalablement dans les Figure 6-7 et Figure 6-8 du chapitre précédent.

```
1  %Context variables
2  array[1..7] of float:BatteryLevel;
3  int:Temperature;
4  float:Brightness;
5  float:minBrightness;
6  float:opBrightness;
7  float:TankLevel;
8  float:MinTankWater;
9  float:OpTankWater;
10 float:WaterConsumption;
11 0..1:Rain;
12 float:EConsumption;
13 float:OpConsumption;
14 0..1:PowerFailure;
15 int:Month;
16 float:HumidityLevel;
17 float:NormalHumidity;
```

Figure 7-2 : Variables du contexte telles que générées dans Minizinc

```
1  %Declaration of variables
2  var 0..1:Fleet;
3  var 0..1:Actuator;
4  array[1..9] of var 0..1:Sprinkler;
5  array[1..9] of var 0..1:Memory;
6  array[1..9] of var {0, 45, 90, 180, 360}:Rotation;
7  array[1..9] of var 0..1:Head;
8  array[1..9] of var 0..60:PressureSp;
9  array[1..9] of var 0..1:Rotor;
10 array[1..9] of var 0..1:Spray;
11 var 0..1:CenterPivor;
12 var {0, 114,232}:EmitterCP;
13 var 0..60:PressureCP;
14 var 0..1:SprinklerCP;
15 var 0..1:Dripline;
16 var {0, 114,232}:Emitter;
17 var 0..60:PressureDr;
18 var 0..1:Humidifier;
19 var 0..1:Rooftop;
20 var 0..1:Position;
21 var 0..1:OUT_open;
22 var 0..1:OUT_close;
23 var 0..1:Partial;
24 var 0..1:Roof;
25 var 0..1:Shader;
26 var 0..1:Glass;
27 array[1..5] of var 0..1:LightBulb;
28 var 0..1:MistDispenser;
29 var 0..1:MistTemp;
30 var 0..1:PressureMD;
31 var 0..1:Nozzle;
32 array[1..4] of var 0..1:AirConditionner;
```

Figure 7-3 : Variables du système telles que générées dans Minizinc

Les variables booléennes, les énumérations, les variables de domaine et les multi-instances sont respectivement traduites comme il est illustré dans les lignes 2, 16, 13, 4

de la Figure 7-3. Similairement, les variables numériques sont traduites tel qu'illustré par les lignes 3 et 4 de la Figure 7-2. Les variables de contexte et paramétriques sont instanciées à travers un modèle de perception qui définit leurs valeurs.

Ensuite, toutes les contraintes définies en dehors des états, sont considérées externes, c'est à dire qu'elles proviennent des modèles de variabilité de domaine externes. Elles sont transformées en contraintes flexibles afin d'éviter toute contradiction avec les contraintes récemment intégrées, et garantir ainsi un modèle valide. La Figure 7-4 représente la transformation des contraintes externes en contraintes flexibles.

```
1  %Flexing external constraints
2  var 0..1: Flex1;
3  constraint Flex1 = 1 <-> (Fleet=Communication);
4  var 0..1: Flex2;
5  constraint Flex2 = 1 <-> (Fleet=Actuator);
6  var 0..1: Flex3;
7  constraint Flex3 = 1 <-> (Fleet=Sensor);
8  constraint Flex22 = 1 <-> (Actuator≥Humidifier);
9  var 0..1: Flex23;
10 constraint Flex23 = 1 <-> (Actuator≥LightBulb[1] /\ ...
       Actuator≥LightBulb[2] /\ Actuator≥LightBulb[3] /\ ...
       Actuator≥LightBulb[4] /\ Actuator≥LightBulb[5] /\ ...
       Actuator=1 -> ((LightBulb[1]+ LightBulb[2]+ LightBulb[3]+ ...
       LightBulb[4]+ LightBulb[5])>0 /\ (LightBulb[1]+ ...
       LightBulb[2]+ LightBulb[3]+ LightBulb[4]+ LightBulb[5])<5));
11 var 0..1: Flex74;
12 constraint Flex74 = 1 <-> (Humidifier>0 -> ...
       (Air[1]+Air[2]+Air[3]+Air[4]+Air[5]+Air[6]+Air[7]>0));
13 var 0..1: Flex75;
14 constraint Flex75 = 1 <-> (OUT_open * (LightBulb[1]+ ...
       LightBulb[2]+ LightBulb[3]+ LightBulb[4]+ LightBulb[5]) =0);
```

Figure 7-4: Contraintes externes flexibles telles que générées dans Minizinc

La contrainte de la ligne 10 dans la figure ci-dessus décrit la relation de groupe [1..5] imposée sur le groupe des ampoules. Cependant, plus tard, selon les exigences de Maria, cette relation deviendra [0..3]. Ainsi, et afin de ne pas être en contradiction avec cette première, la contrainte réifiée Flex23 =1 est rajoutée.

Les états-contraintes simples, les états composites ainsi que les niveaux de préoccupation sont définis comme variables booléennes, décrites dans la Figure 7-5. Leur hiérarchie, elle, est décrite sous forme de contraintes tel qu'illustré dans la Figure 7-6. Chaque niveau supérieur est transformé en contrainte réifiée. Lorsque ses états supérieurs sont actifs, les contraintes qui leur correspondent sont valides. Les contraintes que ces dernières réifient le sont aussi par conséquent, à savoir les sous états.

```
1  %Declaration of the Statemachine hierachy
2  var 0..1:SS;
3  var 0..1:global_SS;
4  var 0..1:global_SS_state;
5  var 0..1:irrigation_SS;
6  var 0..1:automatic_SS;
7  var 0..1:automatic_CL_SS;
8  ...
9  var 0..1:NoE;
10 var 0..1:global_NoE;
11 var 0..1:global_NoE_state;
12 var 0..1:irrigation_NoE;
13 ...
```

Figure 7-5: Déclaration des variables d'états et de niveaux de préoccupation

```
1  %Constraints on the Statemachine hierachy
2  constraint AW = 1 <-> global_AW = 1;
3  constraint global_AW = global_AW_state ;
4  constraint AW = 1 <-> irrigation_AW = 1;
5  constraint irrigation_AW = Natural_Irrigation_AW_state + ...
       automatic_AW  ;
6  constraint automatic_AW = 1 <-> automatic_CL_AW = 1;
7  constraint automatic_CL_AW = rapidIrrigation_AW + ...
       slowIrrigation_AW  ;
8  constraint automatic_CL_AW =  automatic_AW  * ( ...
       automatic_CL_AW );
9  constraint AW = 1 <-> heating_AW = 1;
10 constraint heating_AW = normal_heating_AW_state + ...
       high_heating_AW_state + open_air_AW_state + ...
       extra_heating_AW_state  ;
11 ...
12 constraint SS = 1 <-> waterSource_SS = 1;
13 constraint waterSource_SS = efficientWater_SS_state + ...
       durableWater_SS_state  ;
14 constraint global_SS + irrigation_SS + heating_SS + ...
       moitSensing_SS + brightness_SS + waterSource_SS  =  SS  * ...
       ( global_SS + irrigation_SS + heating_SS + moitSensing_SS ...
       + brightness_SS + waterSource_SS  );
15 constraint NoE = 1 <-> global_NoE = 1;
16 constraint global_NoE = global_NoE_state ;
17 constraint NoE = 1 <-> irrigation_NoE = 1;
18 constraint irrigation_NoE = Natural_Irrigation_NoE_state ;
19 constraint NoE = 1 <-> heating_NoE = 1;
20 constraint heating_NoE = natural_heating_NoE_state + ...
       automatic_heating_NoE_state  ;
21 constraint NoE = 1 <-> moitSensing_NoE = 1;
22 constraint moitSensing_NoE = survival_moist_NoE_state ;
23 constraint global_NoE + irrigation_NoE + heating_NoE + ...
       moitSensing_NoE  =  NoE  * ( global_NoE + irrigation_NoE ...
       + heating_NoE + moitSensing_NoE  );
24 ...
```

Figure 7-6: Contraintes de hiérarchie telles que générées dans Minizinc

La ligne 2 de la Figure 7-5 illustre la variable SS, correspondant à l'état composite SS du mode Spring_Summer. Similairement, les états contraintes simples irrigation_SS et automatic_SS (lignes 4 et 5) ainsi que le niveau de préoccupation automatic_CL_SS (ligne 7) sont transformés en variables booléennes. Toute la hiérarchie de la machine à états est transformée en contraintes, tel qu'illustré dans la Figure 7-6. Les lignes 2, 4, et 9 illustrent l'activation des niveaux de préoccupation global_AW, irrigation_AW, et heating_AW de l'état composite orthogonal AW, correspondant au mode Autumn_Winter. La ligne 7

illustre le flux d'exécution à l'intérieur du niveau de préoccupation automatic_CL_AW, définit à travers les états séquentiels rapidIrrigation_AW et slowIrrigation_AW.
Les transitions, porteuses de conditions, sont traduites en contraintes réifiées, comme il est illustré par la Figure 7-7. La satisfaction de cette dernière implique l'activation de l'état source qui lui est relié. Le cas échéant, l'état source de la transition n'est pas actif et les contraintes qui en découlent ne doivent pas être satisfaites.

```
1 %Transitions
2 constraint NoE = 1 -> (natural_heating_NoE_state =1 \/ ...
      natural_heating_NoE_state+automatic_heating_NoE_state= 1);
3 constraint NoE = 1 -> (survival_moist_NoE_state =1 );
4 constraint (((Month=10)) /\ (NoE=1 )) -> ...
      automatic_heating_NoE_state = 1;
5 constraint (((Month=4)) /\ (NoE=1 )) -> ...
      natural_heating_NoE_state = 1;
```

Figure 7-7: Transitions telles que générées dans Minizinc

Ainsi, la transition vers le mode coupure d'électricité (NoE) entraine, pour chaque niveau de préoccupation, l'activation de son état initial ou bien d'un des autres états (Lignes 2 et 3) dans le cas où la transition est activée. A partir d'Octobre (Ligne 4), le chauffage est automatisé. Ce dernier est désactivé en mois d'avril, du fait que les températures durant les saisons de printemps et d'été sont plus adéquates à cette plantation. Le Rooftop est en conséquence ouvert (Ligne 5). Finalement, les contraintes définies au niveau de chaque état-contrainte simple sont exprimées par une contrainte réifiée, correspondant au statut de cet état. Ainsi, l'activation de l'état garantit la satisfaction de ses contraintes et vice-versa. La Figure 7-8 illustre ces propos.

```
1 %Constraints
2 constraint automatic_cooling_SS_state = 1 -> ...
      ((AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+ ...
      AirConditionner[4]=1 ) /\(Temp[1]+Temp[2]+Temp[3]+ ...
      Temp[4]=20 /\ Temp[1]*Temp[2]=0 /\ Temp[3]*Temp[4]=0 /\ ...
      Temp[1]*Temp[3]=0 /\ Temp[2]*Temp[4]=0 )/\(Speed[1]+ ...
      Speed[2]+Speed[3]+Speed[4]=3 /\ Speed[1]*Speed[2]=0 /\ ...
      Speed[3]*Speed[4]=0 /\ Speed[1]*Speed[3]=0 /\ ...
      Speed[2]*Speed[4]=0 ) /\(Mode[1]+Mode[2]+Mode[3]+Mode[4] ...
      =2 /\ Mode[1]*Mode[2]=0 /\ Mode[3]*Mode[4]=0 /\ ...
      Mode[1]*Mode[3]=0 /\ Mode[2]*Mode[4]=0 ));
```

Figure 7-8: Contraintes du cas GreenLife, telles que générées dans Minizinc

La figure ci-dessus décrit l'ensemble des contraintes qui doivent être satisfaits lorsque l'état-contrainte simple automatic_Cooling_SS_State de la ligne 2 est actif.
Finalement, le modèle, selon la notation SCT génère un fichier (.mzn), qui correspond au problème de contraintes, composé comme décrit plus haut. Le fichier requiert en entrée

les valeurs des variables de contexte et paramétriques, qui peuvent lui être communiquées par un processus externe au modèle. Le tout, exécuté, génère une configuration. Celle-ci correspond à une affectation de valeur à toutes les variables impliquées, qui garantisse toutes les contraintes fortes et qui maximise toutes les contraintes flexibles dans une situation bien définie.

7.6. Traces de configuration : Le cas de la solution GreenLife

La spécification formelle d'un système de systèmes dynamique selon l'approche SCT génère en sortie un programme de contraintes dynamique, dans le sens ou les résultats de la résolution du problème de contraintes dépendent d'un ensemble de paramètres dynamiques. Ceux-ci correspondent au contexte d'exécution et d'utilisation du système de système en question. Ainsi, pour chaque valeur des variables de contexte, une ou plusieurs nouvelles configurations valides peuvent être générées.

Variable	Scénario en mode AW [5]	Scénario en mode SS [6]
Rain	1	0
Month	1	8
PowerFailure	0	0
BatteryLevel	[100,88,95,73,69,73,88]	[40,88,95,53,39,30,8]
Brightness	763	888
opBrightness (lux)	700	700
Temperature	10	12
HumidityLevel	54	24
NormalHumidity	80	80
TankLevel (l)	261	50
OpTankWater (l)	300	300
WaterConsumption	247	150
OpConsumption (Kw)	100	100
EConsumption (Kw)	95	59
minBrightness (lux)	300	300
MinTankWater (l)	100	100

Tableau 7-1: Exemples de variables de contexte

[5] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/ThesisFleetFirst.dzn
[6] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/ThesisFleetSecond.dzn

7.6.1.Scénario aléatoire en mode Autumn_Winter (AW)

Les valeurs illustratives présentées dans la première colonne du Tableau 7-1, correspondent au scénario décrit dans la deuxième colonne du Tableau 7-2 Le comportement attendu est présenté et est comparé aux résultats actuels de l'exécution du programme de contraintes, généré à partir du modèle SCT.

Valeurs en entrée	Description	Comportement attendu
`Rain=1;`	Il pleut	Les dispositifs d'irrigation doivent être désactivés ;
`Month=1;`	Le mois de Janvier est un mois d'hiver	Le mode AW doit être activé
`PowerFailure=0;`	Le courant électrique est normal	Il n'y a pas besoin de basculer en mode coupure d'électricité
`BatteryLevel=[100,88,95,73,69,73,88];`	Toutes les batteries des capteurs d'humidité sont chargées	Le nombre de capteurs d'humidité actifs doit être maximisé selon les exigences du mode actif
`Brightness =763; opBrightness=700;`	La luminosité est au-dessus de l'optimal	Les dispositifs de régularisation de luminosité, dans le mode AW, doivent être désactivés
`Temperature=10;`	La température est en dessous de la température requise	Les dispositifs de régularisation de température, dans le mode AW, doivent être activés
`HumidityLevel=54; NormalHumidity=80;`	L'humidité du sol est en dessous de la normale	Étant donné que la toiture est ouverte et que la pluie tombe, aucun dispositif d'irrigation ne doit être activé
`TankLevel=261; OpTankWater=300;`	Le niveau d'eau dans le réservoir est en dessous de la mesure optimale	Le réservoir peut être utilisé comme distributeur d'eau
`WaterConsumption=247; OpConsumption=100;`	La consommation globale d'eau ne dépasse pas la consommation optimale	Si irrigateurs sont sélectionnés, ils doivent fonctionner en mode économie d'eau
`IN_open=1;`	La toiture est ouverte	Seuls les climatiseurs peuvent être utilisés pour réchauffer le champ.

Tableau 7-2: Descriptif du contexte AW et du comportement attendu du SdS_Dy

L'exécution du programme de contraintes, dans le contexte AW, génère la configuration décrite par le graphe de la Figure 7-9, qui correspond en effet aux exigences de Maria.

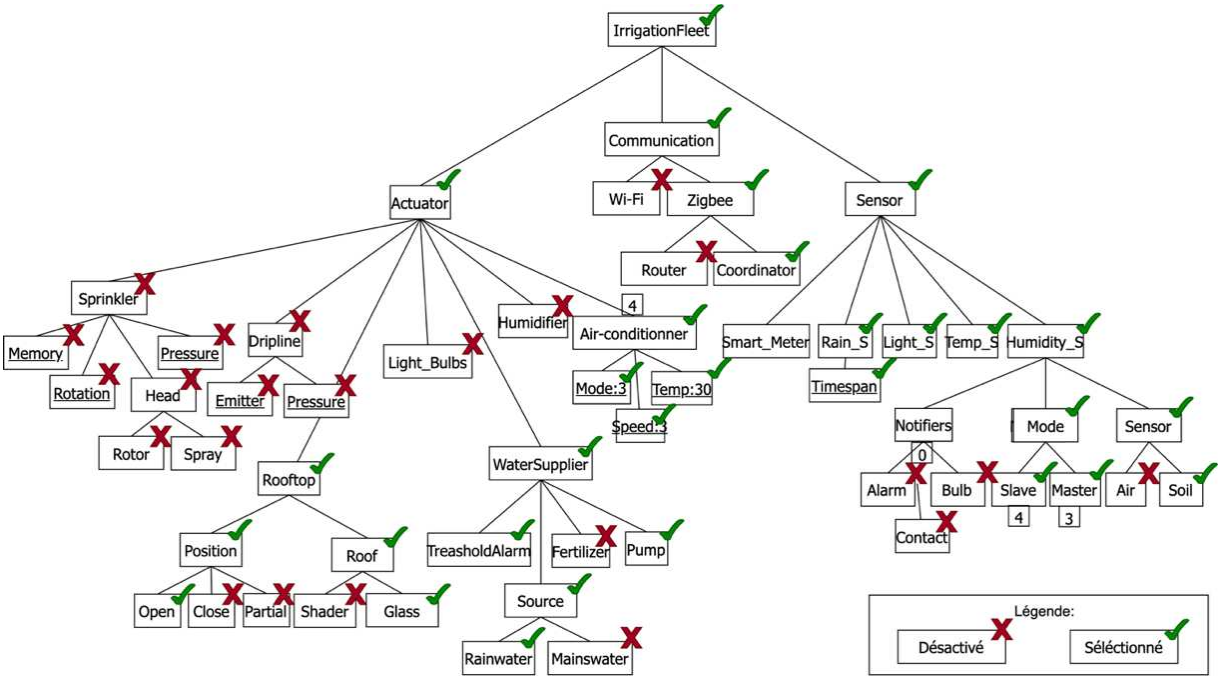

Figure 7-9: Résultat de la configuration pour le contexte AW (Cas Maria)

Par ailleurs, on distingue également les valeurs attribuées aux variables énoncées dans le Figure 7-10, qui déterminent le statut (activé ou désactivé) des états et des niveaux de préoccupation déclarés dans le modèle SCT, pour le cas de Maria.

Ainsi, pour le contexte courant, l'état Natural_Irrigation_AW_state, qui représente les exigences qui doivent être satisfaites pour une irrigation naturelle, est activé, car il pleut. L'état open_air_AW_state est aussi actif pour la même cause. Ensuite, comme les niveaux des batteries des capteurs d'humidité sont tous en dessus de 60%, l'état accurate_moist_AW_state est activé. Finalement, les états efficient_Water_AW_state et natural_brightness_AW_state sont actifs, du fait que le niveau d'eau dans le réservoir dépasse et la luminosité sont en dessus du niveau optimal.

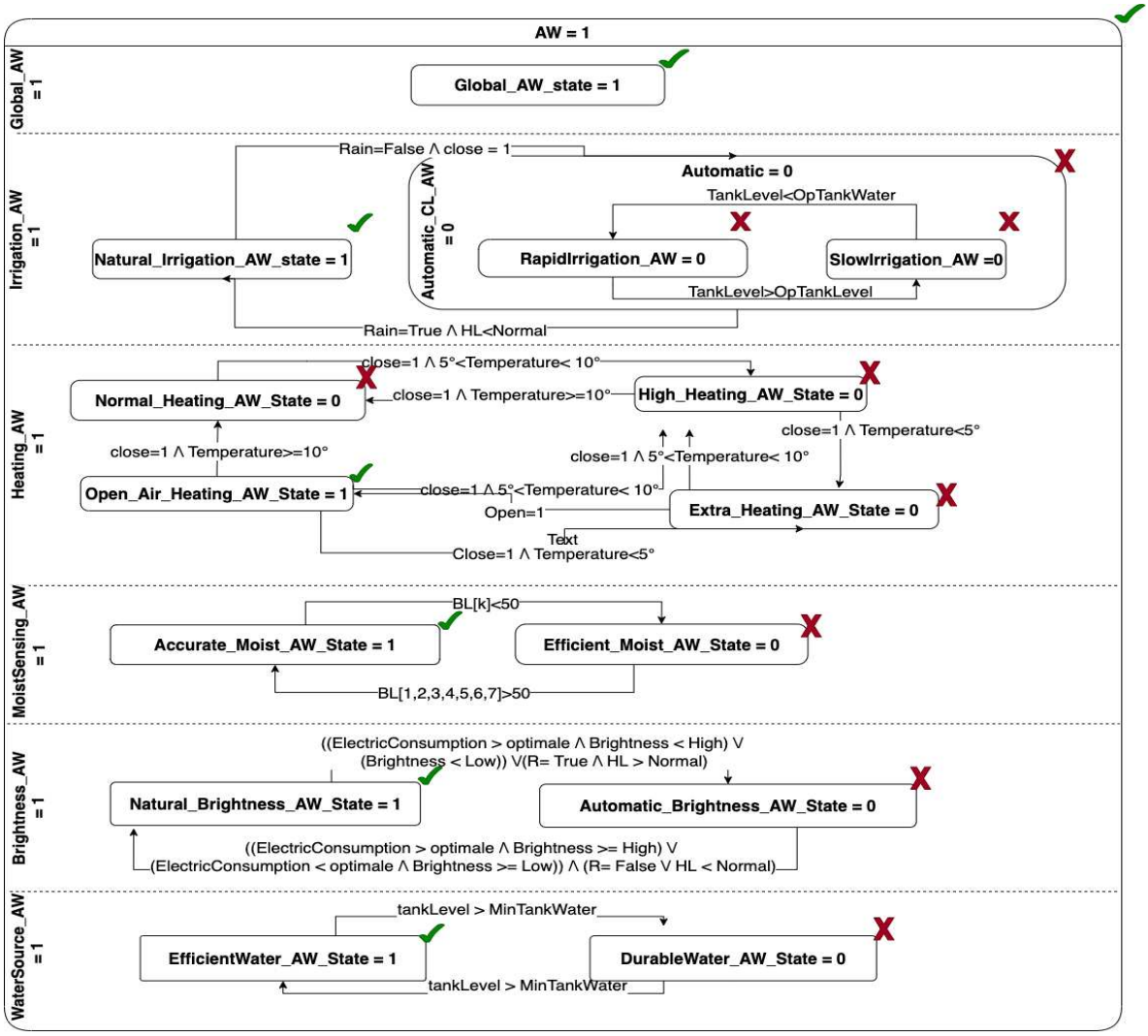

Figure 7-10 : Valeurs des variables d'états et de niveaux de préoccupation (contexte AW)

7.6.2.Scénario aléatoire en mode Spring_Summer (SS)

Comme pour le scénario décrit précédemment, les valeurs de la troisième colonne du Tableau 7-1, correspondent au contexte décrit dans le Tableau 7-3. Elles sont communiquées au problème de contraintes, et le résultat attendu correspond à celui décrit dans la dernière colonne de ce tableau.

Valeurs en entrée	Description	Comportement attendu
Rain=0;	Il ne pleut pas	Les dispositifs d'irrigation doivent être activés ;
Month=8;	Le mois d'Août est un mois d'été	Le mode SS doit être activé
PowerFailure=0;	Le courant électrique est normal	Il n'y a pas besoin de basculer en mode coupure d'électricité
BatteryLevel= [40,88,95,53,39,30,8];	Certaines batteries sont presque déchargées	Le nombre de capteurs d'humidité actifs doit être minimisé selon les exigences de ce mode
Brightness =888; opBrightness=700;	La luminosité est au-dessus de l'optimale	Les dispositifs de régularisation de luminosité, dans le mode SS, doivent être désactivés
Temperature=32;	La température est au-dessus de la température requise	Les dispositifs de régularisation de température, dans le mode SS, doivent être activés
HumidityLevel=24; NormalHumidity=80;	L'humidité du sol est en dessous de la normale	Étant donné que la toiture est fermée et que la pluie ne tombe pas, les dispositifs d'irrigation doivent être activés
TankLevel=50; MinTankWater=100;	Le niveau d'eau dans le réservoir est en dessous de la mesure optimale	Le distributeur de ville est la source d'eau
WaterConsumption=150; OpConsumption=100;	La consommation globale d'eau dépasse la consommation optimale	Si les irrigateurs sont sélectionnés, ils doivent fonctionner en mode économie d'eau
IN_close=1;	La toiture est fermée	L'irrigation automatique peut être activée

Tableau 7-3 : Descriptif du contexte SS et du comportement attendu du SdS_Dy

Le résultat d'exécution du problème de contraintes, ayant en entrée les valeurs propres au contexte SS, est représenté dans la Figure 7-11. Les sprinklers 1, 3, 7 et 9 sont ainsi responsables de l'irrigation, car ils fonctionnent en mode Rotor. Un seul climatiseur est nécessaire pour abaisser la température, celle-ci étant légèrement haute. Le capteur de luminosité est momentanément désactivé, car la luminosité pendant cette saison est généralement très haute.

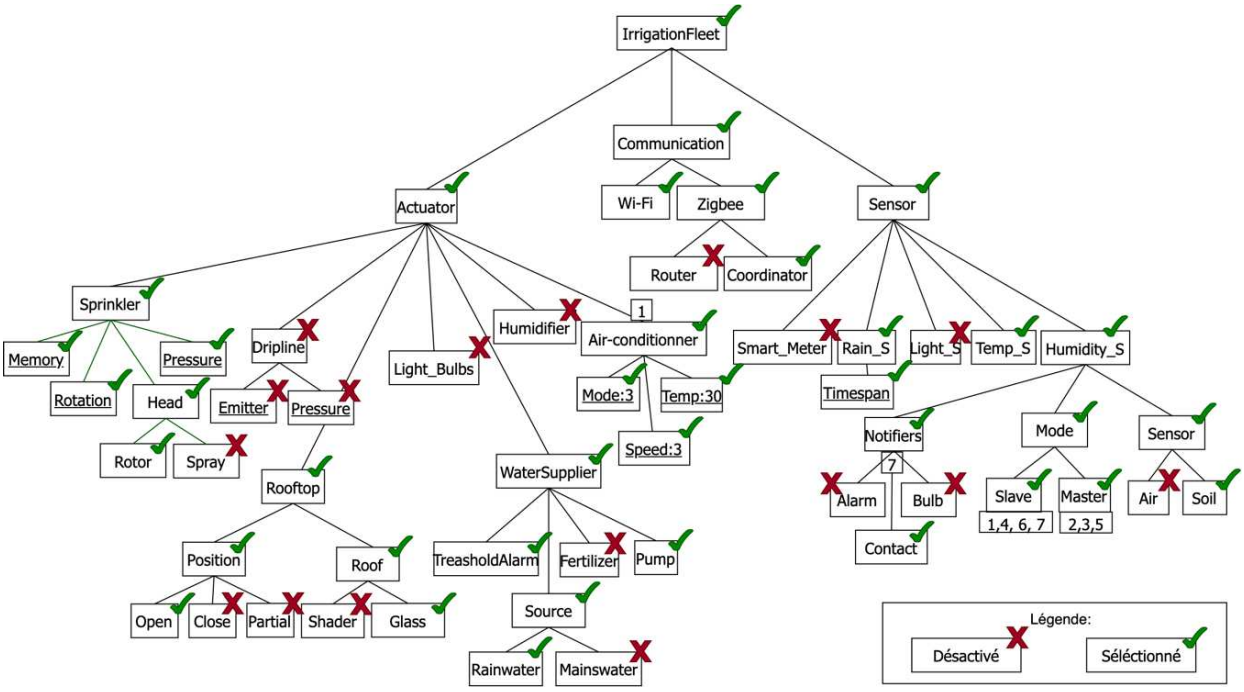

Figure 7-11: Résultat de la configuration pour le contexte SS

Les valeurs des états et niveaux de préoccupation définis dans le modèle SCT, pour le cas du SdS_Dy pour la cliente Maria, suite au contexte SS, sont présentés dans le Figure 7-12. Le mois d'Août indique l'activation de l'état composite SS. Les états des dispositifs d'irrigation (irrigation_SS), de régularisation de température (automatic_cooling_SS_state) et de luminosité(natural_brightness_SS_state) sont actifs, et ce, dans un contexte où il ne pleut pas, le niveau de température dépasse la valeur requise, et la luminosité est inférieure à la valeur optimale. Par ailleurs, efficient_moit_SS_state est actif à la suite du nombre de batterie faible des capteurs d'humidité. Finalement, le niveau d'eau dans le réservoir dépassant le niveau optimal, l'état efficientWater_SS_state est activé par conséquent.

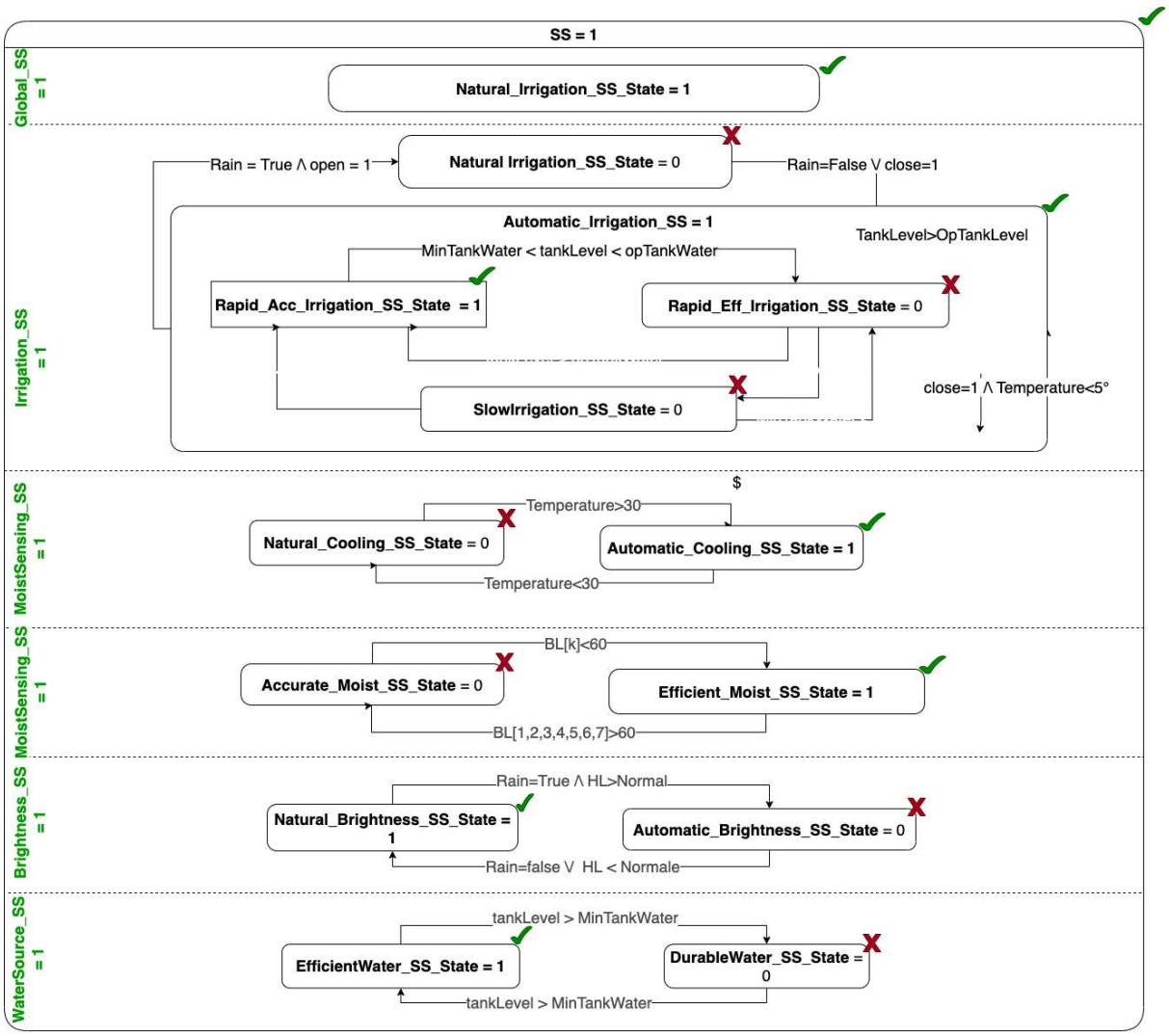

Figure 7-12 : Valeurs des variables d'états et de niveaux de préoccupation (contexte SS)

7.7. Conclusion

Un langage de spécification n'est formel que lorsqu'il offre les mécanismes nécessaires pour traiter automatiquement le document de cette spécification, et ce afin de maintenir un traçabilité tout au long du processus d'ingénierie, pour simuler, tester et valider les exigences avant de passer aux étapes d'ingénierie qui en suivent. Dans cet esprit, ce chapitre présente l'approche élaborée pour automatiser le traitement des modèles SCT proposés dans le chapitre précédent. Parmi les démarches de configuration automatique des lignes de produits, la programmation par contraintes a été retenue. Le chapitre décrit donc, à travers des règles de transformation, comment le modèle SCT est traduit en problème de contraintes dynamiques.

Configurations dynamiques

D'abords les variables du système sont déclarées dans leurs domaines respectifs. Les variables de contexte ainsi que les variables paramétriques du système quant à eux, sont déclarées en tant que paramètres d'entrée au programme de contraintes. Ensuite, les contraintes de domaine qui sont définies en dehors des états sont traduites en contraintes flexibles. L'ensemble des états et niveaux de préoccupations sont déclarées en tant que variables booléennes, et leur hiérarchie est déclarée sous forme de contraintes, en accord avec les règles de transformation définies. Puis, les contraintes définies au niveau des états-contraintes simples sont réifiées par des contraintes appliquées sur l'état-contraintes simple qui leur correspond, de manière à restreindre leur applicabilité au cas où l'état est activé. Et finalement, les transitions sont réifiées par des contraintes représentatives des événements qui les gardent.
Le Framework Xtext est utilisé pour faciliter la spécification de SdS_Dy selon l'approche SCT et produit en sortie un programme de contraintes dont la résolution à un moment (t) présente une ou plusieurs configurations conformes aux exigences et au contexte. Le cas de la cliente Maria, est présenté pour illustrer cette transformation, et évaluer les configurations générées par rapport aux besoins exprimés par cette première.

Chapitre 8 Évaluation

8.1. Introduction

Le langage SCT étend des concepts des modèles machine à états finis (FSM), pour décrire le comportement dynamique des SdS_Dy, conçus comme une ligne de produits logiciels dynamique. De plus, il adopte la sémantique de programmation par contraintes pour générer des problèmes de satisfaction de contraintes. Cette contribution demeure cependant incomplète sans une évaluation qui examine son potentiel en rapport avec les objectifs globaux de cette thèse.

Ce chapitre présente donc une évaluation du langage SCT selon deux approches, d'abord, suivant l'approche Objectif-Questions-Métriques (OQM) (Goal-Question-Metric, GQM en anglais) pour évaluer le pouvoir d'expression, le passage à l'échelle et l'indépendance du domaine, puis, suivant l'approche de la théorie perceptive (Moody 2009) pour évaluer l'efficacité perceptive. Trois cas sont utilisés pour appuyer les différents propos de cette évaluation. Les résultats sont finalement présentés et discutés.

8.2. Approches d'évaluation

Le langage SCT est évalué sur deux volets. D'abord le pouvoir d'expression, le passage à l'échelle et l'indépendance au domaine sont évalués selon une approche But-Questions-Métriques (van Solingen et al. 2002), ensuite, l'efficacité perceptive de la notation SCT est évaluée selon la théorie perceptive de Moody (Moody 2009), adaptée à la représentation textuelle.

8.2.1. Évaluation de la notation SCT selon l'approche Objectif-Questions-Métriques

L'approche But – Questions – Métriques est utilisée pour évaluer trois propriétés du langage de spécification SCT, son pouvoir d'expression, son passage à l'échelle et son indépendance par rapport à un domaine particulier.

- **L'expressivité** fait référence à la capacité du langage à spécifier autant de besoins que nécessaire.
- **Le passage à l'échelle** se traduit par l'aptitude de SCT à modéliser des systèmes de grandes tailles.

- **L'indépendance vis à vis du domaine** détermine le pouvoir du langage à être utilisé dans divers domaines.

Nous avons donc identifié un objectif, traduit par trois questions et cinq métriques connexes.

But : évaluer le pouvoir d'expression, le passage à l'échelle, et l'indépendance de domaine du langage SCT, du point de vue des ingénieurs de domaine, afin d'analyser son efficacité à spécifier des SdS_Dy du monde réel (Achtaich et al. 2019).

- **Q1 :** Le langage de spécification SCT fournit-il les mécanismes nécessaires pour spécifier les exigences diverses des SdS_Dy?
  - **M1 :** Pourcentage (% Ex) d'exigences spécifiées avec succès par les concepts du langage SCT, par rapport à la typologie spécifiée dans le Chapitre 2
- **Q2** : Comment le langage SCT gère-t-il des modèles de grandes tailles ?
  - **M2** : Temps d'exécution (ExT) des systèmes de différentes échelles
  - **M3** : Rapport de dépendance (Cov) entre les temps d'exécution (ExT) et le nombre d'éléments (#E)
- **Q3** : Le langage SCT peut-il spécifier les exigences de systèmes de divers domaines ?
  - **M4** : Nombre de systèmes de domaines d'application différents pouvant être spécifiés avec succès en utilisant la notation SCT, en ce qui concerne la typologie spécifiée dans le Chapitre 2
  - **M5** : Pourcentage d'exigences (% Ex) spécifiées à l'aide de la notation SCT parmi les cas étudiés

8.2.2. Évaluation de la notation SCT selon la théorie perceptive

La théorie perceptive définit un ensemble de principes assurant l'efficacité cognitive des notations. Introduits par Moody dans (Moody 2009), ces principes sont principalement conçus pour l'évaluation des langages de spécification graphiques. Le Framework de cette théorie présente neuf principes, notamment, la clarté sémiotique, la distinction perceptive, la transparence sémantique, la gestion de la complexité, l'intégration cognitive, le pouvoir d'expression visuel, l'ajustement cognitif, l'économie graphique et le double codage.

Le langage SCT peut avoir aussi bien une représentation graphique que textuelle. Le prototype proposé dans cette thèse est sous format textuel, en conséquence, le

Framework de Moody est adapté de manière à s'accommoder à la représentation textuelle du langage évalué. La Figure 8-1 résume l'ensemble des principes retenus, ainsi que les métriques utiles à l'évaluation.

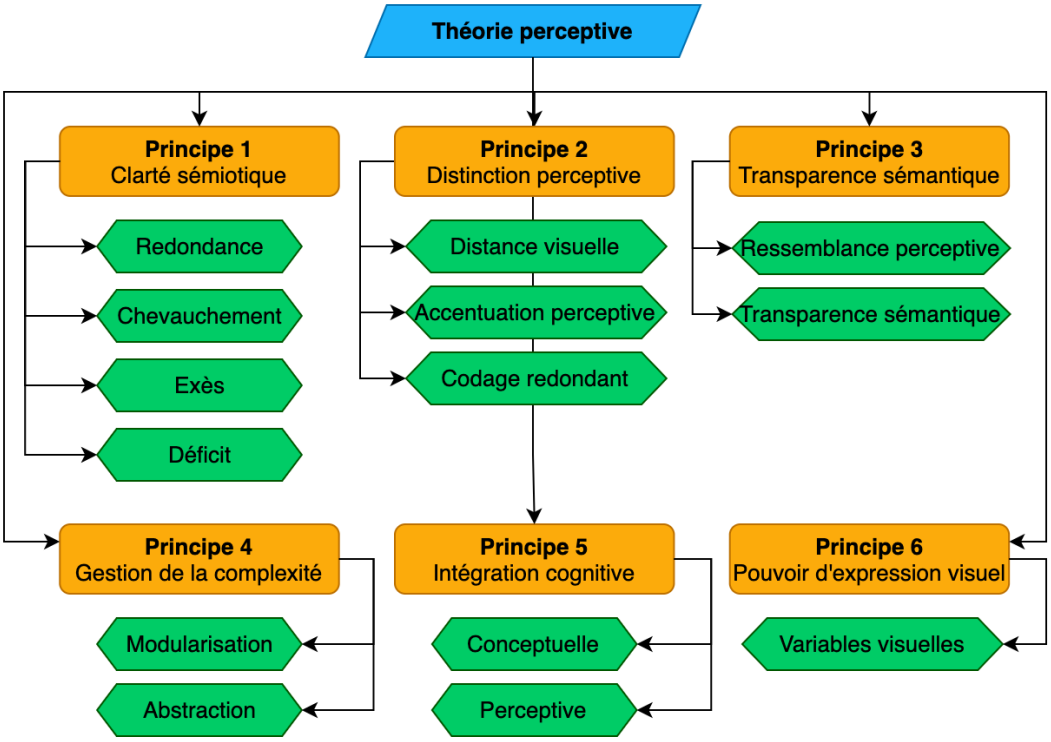

Figure 8-1 : Principes et métriques de la théorie perceptive

8.2.2.1. 1er principe : La clarté sémiotique

La clarté sémiotique fait référence à la capacité d'un langage à avoir une relation une-à-une entre les concepts du méta-modèle du langage et la notation effective. Le cas échéant, le langage est susceptible de manquer de précision, de pouvoir d'expression, et de parcimonie. Quatre anomalies de langage peuvent en résulter ; la **redondance**, le **chevauchement**, **l'excès** et le **déficit**. La première anomalie se produit lorsque plusieurs éléments de la notation sont utilisés pour représenter un concept unique. La deuxième à lieu lorsqu'un élément de la notation est utilisé pour représenter des concepts différents du méta-modèle. La troisième est résultat d'un élément de la notation qui ne représente aucun des concepts du langage. Et finalement, la quatrième anomalie fait référence à la présence de concepts qui ne sont pas interprétés par la notation.

8.2.2.2. 2ème principe : La distinction perceptive

La distinction perceptive fait référence à la facilité avec laquelle une notation est perceptible, afin d'aider dans la compréhension globale des modèles réalisés. Dans le cas d'une notation textuelle, la **distance visuelle, l'accentuation perceptive** et le **codage redondant** sont les caractéristiques retenues. Le premier fait référence à la manière avec laquelle les différents éléments de la notation sont organisés et séparés les uns par rapport aux autres, de manière à ce qu'ils soient distingués visuellement. Les deuxième et troisième propriétés font respectivement référence aux variables visuelles et les combinaisons uniques de celles-ci, qui facilitent cette distinction, et ce pour les différents concepts de la notation.

8.2.2.3. 3ème principe : La transparence sémantique

La propriété de transparence sémantique se rapporte à la mesure avec laquelle le sens des éléments du langage est déduit à son apparence. L'apparence des éléments du langage a ainsi une sémantique immédiate, opaque ou perverse respectivement lorsque le lecteur réussit à inférer son sens correctement, arbitrairement ou différemment. Deux propriétés sont évaluées dans ce sens, la **ressemblance perceptrice** et la **relation sémantiquement transparente**. La première détermine la ressemblance entre la nomenclature choisie et le concept du méta-modèle concerné par celle-ci, et la deuxième évalue la nature des relations entre un ensemble de concepts, et ce, à partir de leur représentation visuelle.

8.2.2.4. 4ème principe : La gestion de complexité

La gestion de la complexité consiste à mettre en place les mécanismes nécessaires pour minimiser la complexité visuelle d'un modèle, particulièrement, dans le cadre de l'ingénierie de systèmes, dans laquelle les modèles sont généralement d'échèle très grande. Contrairement aux modèles graphiques, la complexité dans un modèle textuel est mesurée par le nombre de lignes dans un modèle. Plus celui-ci est grand, plus des limites perceptives et cognitives risquent d'avoir lieu. En effet, plus la complexité du modèle est imposante, plus le modèle devient illisible par le lecteur, et plus sa compréhension devient difficile. Deux caractéristiques entrent en jeu dans la gestion de la complexité d'un modèle,

la **modularisation** et **l'abstraction**. La première encourage la subdivision du modèle en parties perceptibles et cognitives, et la deuxième encourage la séparation de préoccupations de différents niveaux d'abstractions dans des blocs différents.

8.2.2.5. 5ème principe : L'intégration cognitive

Souvent, en ingénierie de systèmes, différents modèles sont utilisés pour représenter les différents points de vue concernés par la modélisation. Ceci est une conséquence de la modularisation. L'intégration cognitive consiste à faciliter l'intégration de l'ensemble de ces modèles, dans le but d'assimiler leurs rôles et leurs contributions dans le modèle global. L'intégration peut être **conceptuelle**, en proposant un modèle global qui incorpore l'ensemble des modèles utilisés ou/et elle peut être **perceptive**, en proposant des mécanismes visuels facilitant le repérage dans le modèle courant et l'orientation vers l'ensemble des modèles existants.

8.2.2.6. 6ème principe : Le pouvoir d'expression visuel

Le pouvoir d'expression visuel est un principe accompli par l'ensemble de variables visuelles utilisées par la notation. Ces variables peuvent faire référence à une position horizontale ou verticale, une taille, une opacité, une couleur, une texture, une forme ou une orientation.

8.3. Les cas d'évaluation

L'évaluation du langage SCT a été réalisée à l'aide de trois cas, dans trois domaines différents. Le premier cas est celui de GreenLife, décrit dans le Chapitre 2 . Il appartient au domaine de l'IdO. Le deuxième cas exploité dans cette évaluation celui du Gridstix (Hughes et al. 2006), des réseaux de capteurs sans fil du domaine des systèmes auto-adaptatifs (Wireless Sensor Netwroks, WSN en anglais). Enfin, le dernier cas est celui des systèmes du train d'atterrissage (Landing Gear System, LGS en anglais) (Boniol and Wiels 2014), du domaine des systèmes complexes. La typologie d'exigences sert de base à l'évaluation du pouvoir d'expression. Le premier cas est donc implémenté, et chaque type d'exigence extrait de ce cas, est évalué, et est associé au concept SCT qui permet sa spécification. Les deux cas restants sont ensuite implémentés. Ensemble, les trois cas

examinent d'une part le passage à l'échelle du langage, étant donné qu'ils sont de tailles distinctes, et d'autre part, l'indépendance par rapport à un domaine particulier. Le 1er cas étant détaillé dans le Chapitre 2 , les deux autres cas sont décrits à la suite.

8.3.1.1. 2ème cas : Gridstix [7]

L'intérêt du cas Gridstix défini par (Hughes et al. 2006) est d'être une référence dans la modélisation de systèmes auto-adaptatifs. Le Gridstix est un réseau de capteurs sans fil (WSN) permettant de détecter et de prévoir les inondations. Chaque nœud du WSN est équipé de capteurs qui détectent la profondeur et le débit d'eau.
D'abord, la communication entre les nœuds peut être établie à l'aide de protocoles « Wi-Fi » ou « Bluetooth ». Le premier est plus tolérant aux pannes mais nécessite plus d'énergie que le second. Ensuite, la transmission peut être établie en utilisant les protocoles de routage « plus court chemin » ou « moins de saut ». Le premier compromet la précision et la tolérance aux pannes, mais consomme moins d'énergie. Finalement, le traitement des données peut être réalisé à l'aide d'algorithmes centralisés ou distribués. La précision est obtenue par le premier type au détriment de l'efficacité énergétique. Le système adapte ces trois aspects en fonction de l'état de la rivière et du niveau de la batterie. L'implémentation du modèle de variabilité de la Figure 8-2 est disponible dans l'annexe B.

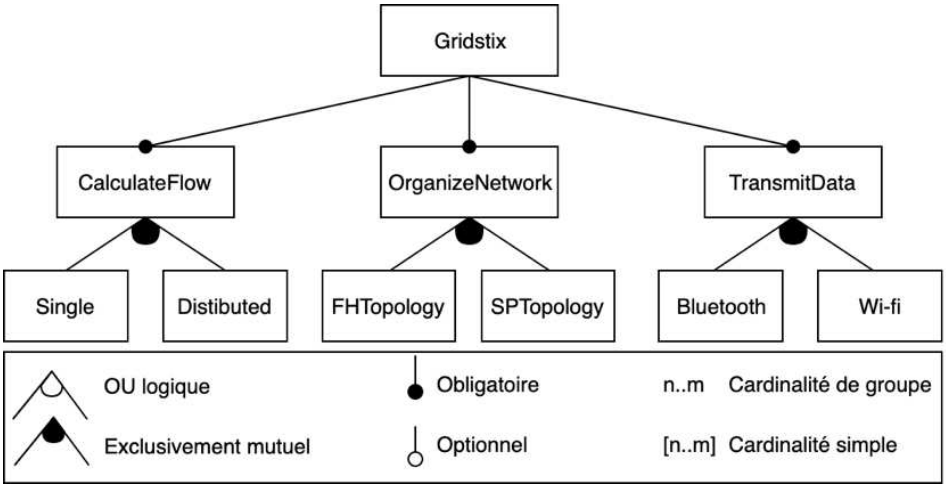

Figure 8-2: Modèle de caractéristiques (Cas 2- Gridstix)

[7] https://github.com/aachtaich/SCT-Models/blob/master/src/Gridstix.sct

8.3.1.2. 3ème Cas : Système de train d'atterrissage [8]

L'intérêt de ce cas, défini par (Boniol and Wiels 2014), est d'être une référence dans la modélisation de systèmes dynamiques. Le système de train d'atterrissage (LGS) introduit un large éventail d'exigences dynamiques, liées à ses différentes propriétés comportementales. Il est contrôlé numériquement en mode nominal et analogique en cas d'urgence. Tel qu'il est présenté, le cas du LGS n'examine pas le mode analogique, mais les paramètres de santé de tous les équipements impliqués dans celui-ci sont définis et sont donc inclus dans cette application. Chacun de ces deux modes a différents sous-états, liés aux états de l'une de ses trois parties, détaillées ci-dessous. Les différentes configurations du LGS partagent des caractéristiques communes, tout en maintenant des distinctions au niveau de parties spécifiques. Ces configurations peuvent donc être considérées comme des produits d'une ligne de produits logiciels dynamiques illustrée dans la Figure 8-3.

L'interface pilote est composée d'une manivelle, qui peut être basculée vers le haut ou vers le bas, pour exécuter respectivement la séquence de rétraction ou d'extension élaborée ci-dessous. Trois lumières informent sur la position des engrenages ; elles sont vertes lorsque les engrenages sont ouverts, oranges en cas de panne, rouges en cas de défaillance et enfin éteintes lorsque les engrenages sont verrouillés en position fermée.

La partie hydraulique comprend :

- Trois ensembles d'atterrissage, chacun composé d'un engrenage, d'une porte et de boîtes de verrouillage qui les verrouillent / déverrouillent.
- Six vérins hydrauliques de commande, responsables de la rentrée / sortie réelle des engrenages et de la fermeture / ouverture des portes, en étant mis sous pression de haut en bas, par leurs électrovannes respectives.
- Cinq électrovannes, règlent la pression sur la partie du circuit hydraulique qui les concerne. Réciproquement, les électrovannes de retrait / extension et d'ouverture / fermeture pressurisent les parties concernées avec la traction / extension des engrenages et l'ouverture / fermeture des portes. Une électrovanne générale alimente l'électrovanne requise à partir du circuit de l'aéronef. Ces électrovannes sont activées par un ordre électrique, provenant de la partie numérique.

[8] https://github.com/aachtaich/SCT-Models/blob/master/src/LGS.sct

- Un total de cinquante-quatre (54) capteurs informe la partie numérique de l'état de l'équipement. Certains de ces capteurs fournissent simultanément trois valeurs concernant la position de la poignée, la position des engrenages, la position des portes, l'état de l'amortisseur et l'état du circuit.

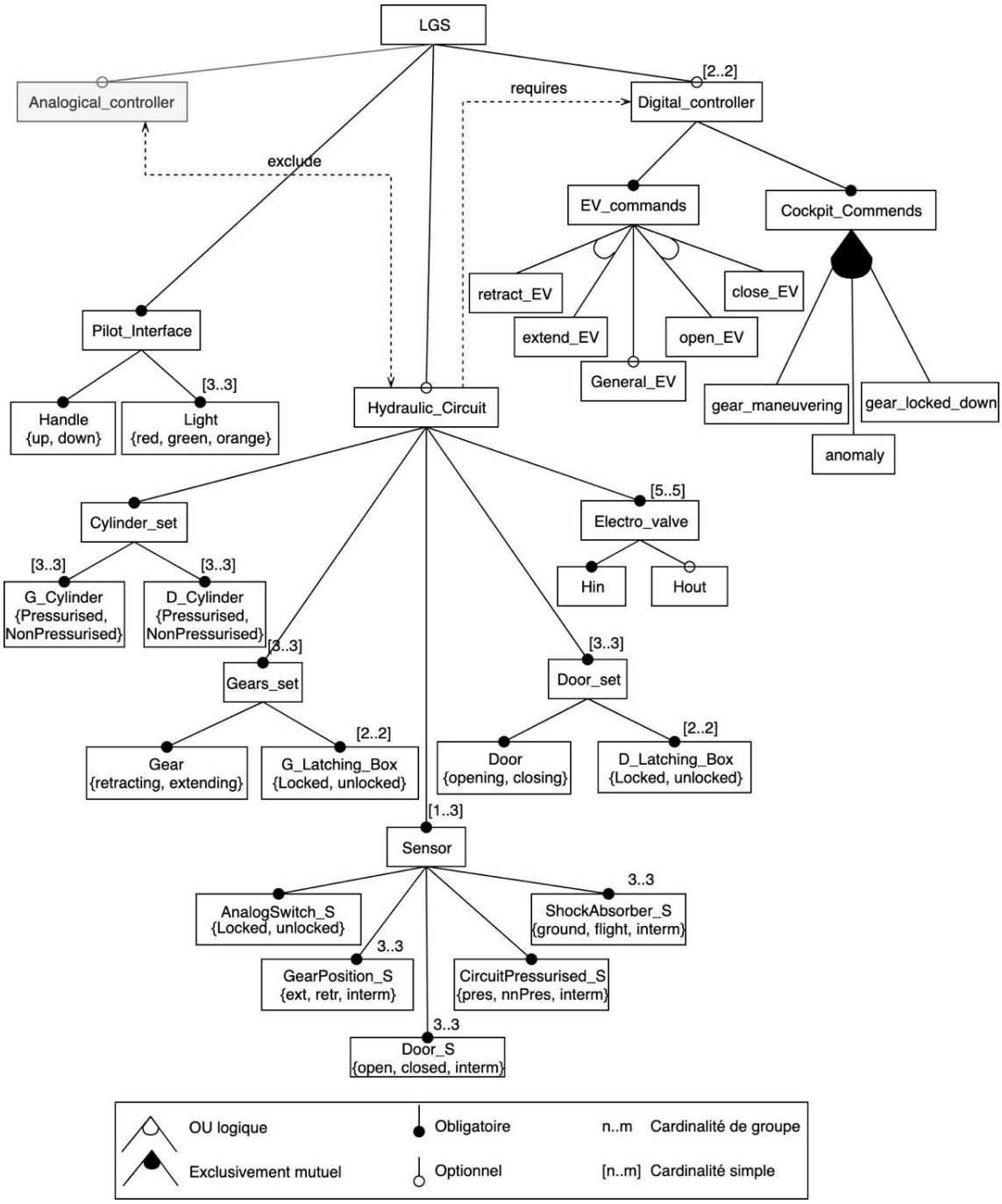

Figure 8-3: Modèle de caractéristiques (Cas 3 - LGS)

La partie numérique est composée de deux modules d'ordinateur exécutant simultanément le même logiciel, permettant de contrôler les engrenages et les portes, de détecter les anomalies et d'informer le pilote de l'état général du système. La partie

numérique reçoit les données des capteurs et, en conséquence, envoie un ordre électrique à une électrovanne qui met ensuite les cylindres connectés sous pression, préformant ainsi l'actionnement nécessaire. Des variables globales sont également produites dans le cockpit.

Le Tableau 8-1récapitule pour chaque cas détaillé plus haut, le nombre d'exigences, de contraintes et de variables.

	# exigences	# variables	# contraintes
Cas 1 : Système d'irrigation [9]	35	388 bool, 306 int	117 bool, 410 int
Cas 2 : Gridstix [10]	17	13 bool, 29 int	4 bool, 28 int
Cas 3 : LGS [11]	20	212 bool, 244 int	89 bool, 304 int

Tableau 8-1: Statistiques des cas d'évaluation

8.4. Résultats de l'évaluation

Neuf propriétés du langage SCT ont été évaluées. Les trois premières, à savoir le pouvoir d'expression, le passage à l'échelle, et l'indépendance par rapport à un domaine spécifique, découlent d'une approche OQM. Les six propriétés qui suivent sont représentatives de l'efficacité perceptive du langage SCT, et sont ainsi les résultats d'une évaluation selon le framework adapté de Moody (Moody 2009).

8.4.1. Pouvoir d'expression

La réponse à la première question (Q1) dépend d'un ensemble d'exigences déduites du premier cas. Le Tableau 8-2 relie chaque type d'exigence au concept du langage SCT qui lui correspond. Les abréviations dans les colonnes du tableau font référence à ces concepts, notamment : ContextVariable (CV), SystemVariable (SV), Constraint (Cst), Event (Ev), Transition (T), simpleConstraintState (SSC), CompositeState (CS) et ConcernLevel (CL).

[9] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/ThesisExample.mzn

[10] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/Gridstix.mzn

[11] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/Landing_Gear_System20.mzn

(✓) fait référence aux concepts permettant de spécifier une exigence.

(⬥) fait référence à des concepts complémentaires pour la spécification d'une exigence donnée.

(x) fait référence aux concepts qui n'interviennent pas dans la spécification de l'exigence en question.

Les exigences fonctionnelles et non-fonctionnelles de domaine sont spécifiées à travers la hiérarchie du modèle SCT, notamment, au moyen des concepts ConcernLevel, simpleConstraintState et compositeState. Concrètement, dans un ConcernLevel, différents états peuvent être exécutés en parallèle, permettant ainsi de satisfaire toutes les exigences fonctionnelles et non-fonctionnelles requises. La variabilité observée dans ce type d'exigences est ensuite décrite dans des sous-états (simpleConstraintState ou compositeState) afin de décrire les différentes exigences d'application qui les opérationnalisent, ainsi que les conditions sous lesquelles elles s'activent. Les exigences de variabilité sont représentées en interconnectant différents types de variables mono ou multi-instanciées (booléennes, entières, réelles ou restreintes dans un domaine de valeurs) par le biais de contraintes. Les exigences d'application et d'adaptation sont principalement réalisées par le biais de contraintes, qui permettent, à travers leur pouvoir d'expression, de décrire des règles représentatives des exigences telles que les exigences de proportionnalité, d'optimisation, de coût ou d'autonomie, et ce dans des simpleConstraintState.

La combinaison état-contraintes/transitions permet la spécification des exigences relaxables, qui garantissent la satisfaction d'un ensemble d'exigences tant qu'une condition est garantie. Lorsque cette condition n'est plus valide, la transition s'active, menant le SdS_Dy vers un groupe d'exigences différents, groupés dans l'état cible de la transition. Tout cela, en atteignant un objectif commun dans le cadre d'un ConcernLevel. Les variables de contexte et système sont déclarées dans leurs domaines respectifs et sont utilisées dans une expression de contraintes pour spécifier les événements qui activent les transitions. Ceux-ci assurent le passage d'un état source à un état cible, décrivant les exigences requises dans un contexte donné. Ceci permet de décrire des exigences contextuelles dans un sens large, et particulièrement, des exigences EvRqs (exigences d'évolution), en décrivant les exigences qui doivent remplacer un ensemble précédent d'exigences dans le cas où celui-ci n'est plus satisfait.

Typologie d'exigences			CV	SV	Cst	Ev	T	SSC	CS	CL	Exemple selon la notation SCT
Exigences de domaine	Segment de marché	Fonctionnelle	x	x	x	x	x	x	x	✓	**concernLevel** Irrigation
		Non Fonctionnelle	x	x	x	x	x	✓	✓	✓	**simpleConstraintState** WaterEfficiency_Mode **compositeState** WaterEfficiency_Mode
	Variabilité	Structure	x	◆	✓	x	x	◆	x	x	SIS = Sensor SIS >= Fertilizer
		Multiplicité	x	✓	✓	x	x	◆	x	x	**Boolean** Spray [**1**,6], **Boolean** Rotor [**1**,6] **simpleConstraintState** Global_State (Irrigator >= Dripline) /\ (2*Irrigator >= Sprinkler[1]+Sprinkler[2]) /\ ((Dripline + (Sprinkler[1]+Sprinkler[2]))=Irrigator*1 \/ (Dripline + (Sprinkler[1] +Sprinkler[2]) = Irrigator*2)) /\((Dripline * (Sprinkler[1] +Sprinkler[2])=0))
		Dépendances	x	◆	✓	x	x	◆	x	x	Slave[1] >0 -> Alarm[1]=0 Sprinkler[1] >= 1 <-> Pressure[1] >= 1
Exigences d' application	Fonctionnelle		◆	◆	✓	x	x	x	x	x	Fleet=WaterSupplier
	Non Fonctionnelle		◆	◆	✓	x	x	x	x	x	WaterEfficiency_Mode=1
	Optimisation		✓	✓	✓	◆	◆	◆	◆	x	minimize(HB[1]+HB[2]+HB[3]+HB[4]+ HB[5])
	Préférence	Booléen	x	✓	✓	x	x	x	x	x	**Boolean** Dripline Dripline = 1
		Paramétrique	✓	✓	✓	◆	◆	◆	◆	x	**Float Param** Consumption **Float Param** Op_consumption **When** Accuracy_mode **if** (Power_Failure = **0** /\ Consumption > Max_consumption) **transition from** …
	Autonomie		◆	◆	✓	x	x	x	x	x	(Sprinkler[1]+Sprinkler[2]+…)* CenterPivor= 0 /\ (Sprinkler[1]+Sprinkler[2] +…) *Dripline=0 /\ (Dripline* CenterPivor=0) /\ (Sprinkler[1]+ Sprinkler[2]+…+ Dripline + CenterPivor >0)tBulb[4]+LightBulb[5] <=5))
	Coût		◆	◆	✓	x	x	x	x	x	Fleet_Budget<=2000
	Proportionnalité		✓	✓	✓	◆	◆	◆	◆	x	(temperature>5)=>HB[1]+HB[2]+HB[3]+ HB[4]+HB[5]=3 /\ (temperature>0 /\ temperature<5)=>HB[1]+HB[2]+HB[3]+ HB[4]+HB[5]=4 /\ (temperature>-5 /\ temperature<0) =>HB[1]+HB[2]+HB[3]+ HB[4]+HB[5]=5
Exigences d' adaptation	Relaxable		x	x	x	◆	✓	✓	✓	✓	**concernLevel** moist_Sensing_SS **simpleConstraintState** acurate_moist_SS_state Master[1]+Master[2]+Master[3]+ Master[4]+Master[5]+Master[6]+ Master[7]=7 **simpleConstraintState** efficient_moist_SS_state **When** SS **if** (BatteryLevel[1]>= 60 /\ BatteryLevel[2]>= 60 /\ BatteryLevel[3]>= 60 /\ BatteryLevel[4]>= 60 /\ BatteryLevel[7]>= 60) **transition from** efficient_moit_SS_state **to** acurate_moist_SS_state
	Optimisation		✓	✓	✓	◆	◆	◆	◆	x	minimize(HB[1]+HB[2]+HB[3]+HB[4]+ HB[5])
	Temporelle		x	x	x	x	x	x	x	x	x
	AwRqs		x	x	x	x	x	x	x	x	x
	EvRqs		x	x	x	◆	◆	✓	✓	x	**When** SS **if** (Accuracy_Mode=0) **transition to** Efficieny _Mode
	Contextuelles		✓	✓	✓	◆	◆	◆	◆	x	**When** Accuracy_mode **if** (HS[1]<50% \/ HS[2]<50% \/ HS[3]<50%) **transition from** Accuracy_Mode **to** Efficiency_Mode

Tableau 8-2: Rapport entre la typologie d'exigences et les concepts SCT

Toutefois, étant donné que tous les concepts sont finalement traduits en un programme de contraintes, les exigences temporelles et AwRqs (exigences de sensibilité) ne sont pas spécifiées de manière efficace et spontanée, car les deux incorporent des concepts liés au temps, notamment l'ordre, la fréquence, ou le séquencement. Le concept de temps constitue toujours une limitation majeure de la programmation par contraintes, ce qui a des répercussions sur le pouvoir d'expression du langage SCT.

Pour conclure, M1=89%. Cette métrique correspond au résultat du rapport entre le nombre d'exigences spécifiées avec succès (=17) et celui des exigences de la typologie (=19). Cette mesure démontre que la notation est capable de spécifier une typologie d'exigences très variée. Elle spécifie en effet les exigences classiques d'ordre fonctionnelles, non-fonctionnelles et contextuelle, mais aussi des exigences plus complexes en raison de la nature des SdS_Dy à être variables dans le temps et dans l'espace. Les types d'exigences qui ne sont pas spécifié de manière efficace sont celles qui relèvent du temps, notamment les exigences temporelles et de sensibilité (AwRq), et ce à cause d'une limitation héritée de la sémantique du langage, à savoir la programmation par contraintes.

8.4.2. Passage à l'échelle

À mesure que les systèmes de systèmes dynamique évoluent, le nombre d'éléments à prendre en compte lors du processus de spécification peut atteindre des milliers voire des millions d'éléments. Le passage à l'échelle est donc un aspect très important, car il détermine la capacité du langage de spécification SCT à gérer des systèmes du monde réel. Le passage à l'échelle, présentée ci-après, fait référence aux temps d'exécution nécessaire pour générer une configuration, tout en effectuant des opérations telles que « Toutes les configurations », « Configuration valide » et « Nombre de configurations ». Lorsqu'une exigence d'optimisation est impliquée, une opération de « maximisation » ou de « minimisation » fait également partie des résultats de l'exécution.

Cas	**Nombre de variables** #variable	**Temps d'exécution de l'opération en secondes** ExT
1er Cas : Solution GreenLife	694	0.014363
2ème Cas : Gridstix	42	0,006042
3ème Cas : LGS	495	0.019769

Tableau 8-3 : Passage à l'échelle du langage SCT

Pour répondre à la question 2 (Q2), les trois cas ont été mis en œuvre. Le Tableau 8-3 présente les temps nécessaires à l'exécution d'une opération de maximisation. Parmi les 3 cas, le cas Gridstix compte le plus petit nombre de variables (42 variables). Il est exécuté en 0,006 secondes. Le cas du LGS, étant 12 fois plus grand que celui-ci (495 variables), est exécuté en seulement 2 fois ce temps. Le 1er cas comprend au plus grand nombre de variables (694 variables), c'est à dire 16 fois le nombre de variables du cas Gridstix, et pourtant, il est aussi exécuté en environ 2 fois le temps que cela avait pris pour ce 2ème cas. On peut conclure qu'un modèle selon la notation SCT, traduit en contraintes dans un problème de satisfaction de contraintes, est une approche qui permet de maintenir une performance satisfaisante. Ce constat est déduit étant donné que les systèmes comptant de 42 à 495 éléments effectuent des opérations diverses en quelques millisecondes.
La covariance des temps d'exécution des 3 cas, pour des nombres de variables respectivement égales à 920, 42 et 495 est de `M3=covariance(#variable, ExT)=1,1'`. Ce résultat faible (≈ 1) confirme un degré d'indépendance fort entre le nombre de variables d'un modèle SCT et le temps de son exécution.

8.4.3. Indépendance du domaine

Pour répondre à la question 3 (Q3), les trois cas ont été mis en œuvre. La métrique M4 est donc égale à (3 Systèmes spécifiés / 3 cas). Il convient cependant de rappeler que les exigences qui relèvent du temps ne sont pas spécifiées de manière efficace.
Ensuite, afin de déterminer la valeur de M5, un scénario hypothétique est présenté et réalisé. Il correspond à un contexte (C_{ni}), avec la reconfiguration attendue. Pour chacun de ces scénarios, la configuration générée par le modèle est comparée à la configuration théorique, prouvant ainsi la capacité du langage (ou pas), à spécifier les exigences des cas des domaines différents.

8.4.3.1. Cas 1 : La solution GreenLife

Contexte (Cn1)[12] : Le contexte décrit un scénario pendant le mois de Janvier, où les niveaux de batterie des capteurs d'humidité sont respectivement de

[12] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/ThesisFleetFirst.dzn

100,88,95,73,69,73,88. Il pleut, le niveau d'humidité du sol est de 54 alors que la normal devrait être 80 et la température est de 10°.

Configuration générée : Le Tableau 8-4 décrit la configuration de quelques dispositifs du système d'irrigation, dans le contexte Cn1.

Éléments du système	Configuration générée
Dripline	0
Sprinkler	[0, 0, 0, 0, 0, 0, 0, 0, 0]
Rooftop	1
AirConditionner	[1, 1, 1, 1]
AirMode	[3, 3, 3, 3]
Temp	[30, 30, 30, 30]
Speed	[3, 3, 3, 3]
HS	[1, 0, 1, 0, 0, 1, 0]

Tableau 8-4: Configuration générée par SCT pout un contexte Cn1

Bien que tous les capteurs d'humidité soient bien chargés, 3 parmi les 7 capteurs sont fonctionnelles, et les autres sont mis en veuille prolongée afin de garantir une durabilité de service pendant ces saisons moins ensoleillées. Ensuite, puisqu'il pleut, et que l'humidité du sol est en dessous de la mesure normale, le système doit ouvrir le toit, et désactiver les autres dispositifs d'irrigation afin d'économiser la consommation d'eau. La température est très basse par rapport à la température idéale pour la plantation qui est de 30°. Les climatiseurs sont activés, en puissance maximale, en mode chauffage et une température souhaitée de 30°.

8.4.3.2. Cas 2 : GridStix

Contexte (Cn2) [13] **:** le niveau de la batterie est normal. L'état de la rivière est une problématique d'urgence.

Configuration générée : Le Tableau 8-5 décrit les résultats du problème de contraintes généré depuis le modèle SCT propre au Gridstix.

Gridstix devrait optimiser la tolérance aux pannes et la précision, au détriment de l'efficacité énergétique. Ainsi, les services les plus efficaces, bien qu'ils soient consommateurs d'énergie, devrait être prioritaires.

[13] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/GridstixData.dzn

Éléments du système	Configuration attendue
WiFi	1
Bluetooth	0
SPTopology	0
FHTopology	1
DistributedProcessing	1
SingleNodeProcessing	0

Tableau 8-5: Configuration générée par SCT pour un contexte Cn2

En effet, et comme décrit dans le tableau ci-dessus, la configuration actuelle compte le Wi-Fi pour établir la communication avec les nœuds du réseau, la topologie FH pour la transmission des données et l'algorithme de traitement distribué.

8.4.3.3. Cas 3 : Système de train d'atterrissage

Contexte (Cn3)[14]: le pilote pousse la manivelle vers le bas (Position 2) déclenchant ainsi une séquence d'extension du train d'atterrissage. Aucune anomalie n'a été détectée.
Configuration générée : Le Tableau 8-6 résume le comportement résultant de la fonction du système d'atterrisseur en mode analogique, tel que généré par le problème de contrainte généré du modèle SCT propre au système LGS.

Le LGS devrait fonctionner en mode nominal (mode normal) et la séquence d'extension du train d'atterrissage est supposée avoir lieu. Nous considérons chaque tâche dans une séquence comme une configuration partielle. Par conséquent, pour réaliser la séquence, une série de reconfigurations partielles est effectuée. Le résultat de chacune de ces dernières est aussi l'entrée de la suivante.
Ainsi, et comme présenté dans le tableau ci-dessus, le résultat de l'entrée IN1 qui correspond à un capteur de manivelle en mode extension (Cfg0⊥4), est une autre reconfiguration (reconfig1). Reconfig1 correspond à une valve générale stimulée (ReCfg1⊥15) [15]. Cette configuration partielle est l'entrée (IN2) pour la prochaine reconfiguration (reconfig2), qui elle correspond à la stimulation de la valve d'ouverture des portes des trains d'atterrissage (ReCfg2⊥17)[16]. Et ainsi de suite, et en harmonie avec

[14] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/lgs_data_s2.dzn
[15] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/lgs_data_s3.dzn
[16] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/lgs_data_s4.dzn

les exigences d'une séquence d'extension, les valves d'extension de trains d'atterrissage sont stimulé (ReCfg3⊥19)[17], un fois étendue, les valves de fermeture de porte sont stimulée à leur tour (ReCfg4⊥16)[18]. A la fin de cette séquence (ReCfg6)[19], la valve générale n'est plus stimulée, les train d'atterrissage sont étendus, les porte sont refermés, et la manivelle est remise en position neutre.

	Éléments du système	Configuration attendue						
1	Input/ouput	IN_1	OUT_1/IN_2	OUT_2/IN_3	OUT_3/IN_4	OUT_4/IN_5	OUT_5/IN_6	OUT_6/IN_7
2	Séquence de reconfigurations	Cfg0	ReCfg1	ReCfg2	ReCfg3	ReCfg4	ReCfg5	ReCfg6
3	Handle	2	2	2	2	2	0	0
4	Handle_Sensor	→ **[2,2,2]**	[2,2,2]	[2,2,2]	[2,2,2]	[2,2,2]	[2,2,2]	**[2,2,2]**
5	G_cylinders	1	1	1	2	2	2	2
6	D_cylinders	1	1	2	2	1	1	1
7	f_gearPosition_Sensor	[2,2,2]	[2,2,2]	[2,2,2]	→**[1,1,1]**	[1,1,1]	→**[1,1,1]**	[1,1,1]
8	r_gearPosition_Sensor	[2,2,2]	[2,2,2]	[2,2,2]	→**[1,1,1]**	[1,1,1]	→**[1,1,1]**	[1,1,1]
9	l_gearPosition_Sensor	[2,2,2]	[2,2,2]	[2,2,2]	→**[1,1,1]**	[1,1,1]	→**[1,1,1]**	[1,1,1]
10	f_door_Sensor	[1,1,1]	[1,1,1]	→**[2,2,2]**	[2,2,2]	→**[2,2,2]**	→**[1,1,1]**	[1,1,1]
11	r_door_Sensor	[1,1,1]	[1,1,1]	→**[2,2,2]**	[2,2,2]	→**[2,2,2]**	→**[1,1,1]**	[1,1,1]
12	l_door_Sensor	[1,1,1]	[1,1,1]	→**[2,2,2]**	[2,2,2]	→**[2,2,2]**	→**[1,1,1]**	[1,1,1]
13	Hin	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]
14	Hout	[0,0,0,0,0]	[0,0,0,0,0]	→**[1,0,1,0,0]**	[1,0,1,0,1]	[1,1,0,0,0]	**[1,0,0,0,0]**→	**[0,0,0,0,0]**
15	General_EV	**[0,0]**→	→**[1,1]**	[1,1]	[1,1]	[1,1]	**[1,1]**→	**[0,0]**
16	close_EV	[0,0]	[0,0]	[0,0]	**[0,0]**→	**[1,1]**→	→**[0,0]**	[0,0]
17	open_EV	[0,0]	**[0,0]**→	**[1,1]**	[1,1]	**[0,0]**	→[0,0]	[0,0]
18	retract_EV	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	→[0,0]	[0,0]
19	extend_EV	[0,0]	[0,0]	**[0,0]**→	**[1,1]**→	**[0,0]**	→[0,0]	[0,0]
20	anomaly	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]	[0,0]
21	Time_to_close_switch	0	1	1	1	1	1	0

Tableau 8-6: Configuration générée par SCT pour un contexte Cn3

Les événements importants pour chaque scénario de la séquence (INi) sont placés après le symbole (→). Les éléments concernés par la reconfiguration sont mis en évidence comme suit (EtatAvantConfig → EtatAprèsConfig).

Au bout du compte, les exigences des trois cas ont été spécifiées avec succès. sauf lorsqu'il s'agit, partiellement ou globalement, d'exigences liées au temps. Ces dernières n'ont pas

[17] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/lgs_data_s7.dzn

[18] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/lgs_data_s11.dzn

[19] https://github.com/aachtaich/SCT-Models/blob/master/src-gen/lgs_data_s17.dzn

pu être entièrement spécifiées en raison des limitations du paradigme de programmation par contraintes. Néanmoins, il sont bien spécifiés pas des moyens moins précis, mais qui assure leur prise en charge. En effet, les problèmes liés à la configuration par étapes, où l'ordre avec lequel le problème doit être résolu est pertinent —comme il est le cas de la séquence d'extension détaillée à travers le cas du système LGS, sont traités en introduisant des configurations intermédiaires, présentés dans les colonnes du Tableau 8-6. De plus, les reconfigurations planifiées, où les actions sont effectuées avant / après / pendant des temps spécifiques —comme il est les cas d'une nouvelle saison dans le cas du système d'irrigation, sont gérées par des variables booléennes supposées vraies avant / après / pendant les temps spécifiés.

Finalement, pour calculer le nombre d'exigences spécifiées avec succès pour les trois cas, nous décomposons leur nombre. Premièrement, pour les 35 exigences exprimées dans le cas du SdS_Dy d'irrigation, 32 sont entièrement spécifiées et 3 le sont partiellement car elles décrivent un comportement lié au temps. Deuxièmement, les 17 exigences décrites dans le cas GridStix sont pleinement satisfaites. Enfin, parmi les 11 exigences qui résument le comportement général attendu du système LGS, 9 exigences étaient entièrement spécifiées et 2 le sont partiellement, dans la mesure où elles sont liées au temps. Les dix (10) conditions requises pour le mode de défaillance étaient cependant toutes spécifiées partiellement, puisqu'elles décrivent toutes le comportement résultat d'une expiration d'un délai d'attente. Nous attribuons la note 1 à l'exigence entièrement spécifiée. Cependant, les exigences dépendantes du temps ont une note de 0,5, car elles sont partiellement spécifiées. Par conséquent, le nombre total d'exigences spécifiées est

$$M5 = 68.5/75$$

8.4.4.La clarté sémiotique

La Figure 8-4 présente (à gauche) les concepts du méta-modèle du langage SCT, ainsi que les différents éléments du langage implémenté (à droite).

On distingue clairement les homonymes « system variables » et « context variables », qui sont représentés par l'élément « variable », ainsi que les homonymes « event » et « constraint » qui sont représentés en tant que « constraint ». Bien que les variables de contexte soient spécifiées avec le terminal « Param », il n'y a aucune règle syntaxique qui gère cette distinction. En outre, les deux concepts « event » et « constraint » sont des

expressions mathématiques logiques qui suivent la même syntaxe, d'où l'usage du même élément de la notation SCT pour représenter les deux. Afin de remédier à cela, nous envisageons la représentation des contraintes sous format textuel dans les prochaines versions du langage. De ce fait, la contrainte `Sensor >= Air_Humidity_Sensor` deviendra plutôt `Air_Humidity_Sensor is optional.`

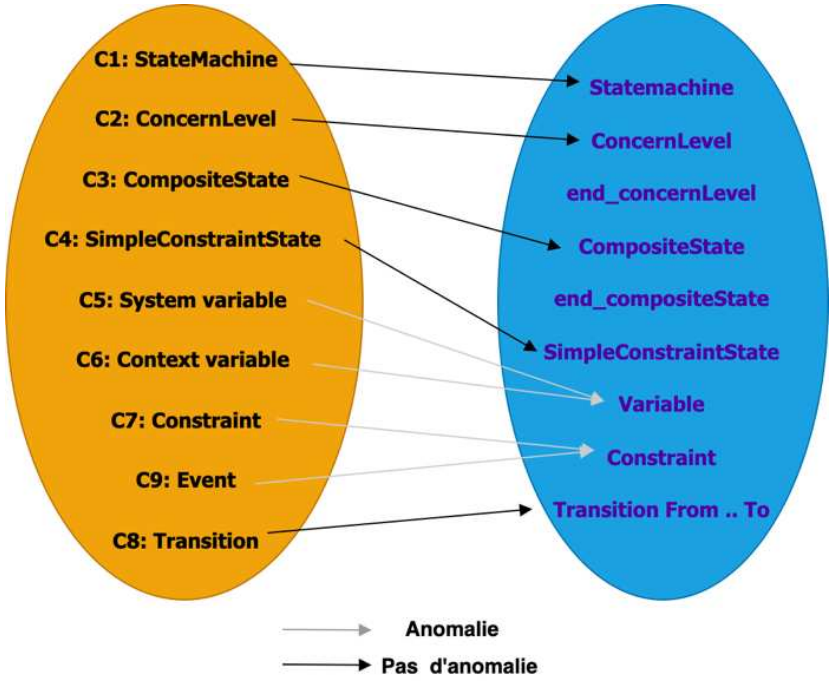

Figure 8-4 : Mapping entre les concepts du méta modèle et ceux de la notation SCT

8.4.5.La distinction perceptive

On distingue deux aspects du langage SCT, qui garantissent une distinction perceptive de l'ordre de la distance visuelle ; notamment, le groupement de concepts analogues et les tabulations, tel qu'illustré dans la Figure 8-5. En effet, la déclaration des variables est faite en amont tout au début du modèle selon la notation SCT, ensuite les états, de nature `SimpleConstraintState` et `CompositeState` sont déclarés et définis, et finalement, les transitions sont énoncées en fin du modèle pour définir les états successifs, et ce par le biais d'évènements déclencheurs. Cette séparation permet au constructeur de modèle de faire la part des choses et au lecteur de repérer facilement les aspects du langage qui l'intéressent. Cette séparation représente implicitement les différents modèles du Framework SPL4DSoS implémentés dans le langage SCT.

```
//Déclaration des variables
78  Float Param opBrightness
79  Float Param TankLevel
80  Float Param MinTankWater
81  Float Param OpTankWater
82  Float Param WaterConsumption
83  Boolean Param Rain
84  Float Param EConsumption
    ...
//Déclaration des états
281     concernLevel brightneAW_AW
282         simpleConstraintState natural_brightness_AW_state
283             OUT_open =1
284         simpleConstraintState automatic_brightness_AW_state
285             OUT_close =1
286             LightBulb[1]+LightBulb[2]+LightBulb[3]+LightBulb[4]+LightBulb[5]>0
287     end_concernLevel brightneAW_AW
    ...
//Déclaration des transitions
484  When SS if (PowerFailure=1) transition from SS to NoE
485  When SS if (Month>=10 \/ Month<4) transition from SS to AW
```

Figure 8-5 : Échantillon illustratif d'un modèle SCT

Par ailleurs, les tabulations sont utiles dans la mise en avant de la hiérarchie des états, afin d'indiquer les états de plus haut et ceux de plus bas niveaux, ainsi que leur appartenance au ConcernLevels. Cependant, elles ne sont pas ajoutées automatiquement, et dépendent entièrement du bon sens du réalisateur du modèle. Bien que certaines variables visuelles permettent de détecter rapidement les concepts clés du langage SCT, telles que la couleur et la texture, comme il est le cas du ConcernLevel, SimpleConstraintState ou CompositeState par exemple, il n'y a aucune combinaison de variables visuelles qui représente de manière exclusive, un élément du langage. Il est donc visuellement difficile de cibler un concept particulier. Finalement, plusieurs éléments de repérage sont effectivement combinés pour mettre en valeur certains concepts et relations du langage, notamment, la couleur, le gras et la position.

8.4.6. La transparence sémantique

La grammaire du langage SCT est assez simple, dans le sens où elle maintient la nomenclature des concepts du langage dans sa notation. Autrement dit, les états-contraintes simples sont déclarés en tant que SimpleConstraintState, pareil pour les niveaux de préoccupation, repérables par leur nom ; ConcernLevel, et ainsi de suite pour les états composites et pour la déclaration des types de variables. Néanmoins, les contraintes et les événements introduisent une confusion. Bien qu'ils soient déclarés dans

des contextes différents, les contraintes dans des états-contraintes simples, et les événements dans une transition, ils sont représentés à travers la même syntaxe, à savoir une expression logique. Par conséquent, perçus hors ce contexte, il devient difficile de détecter le concept concerné par l'expression en question.
Sur un autre volet, on distingue dans le langage SCT quarte types de relations : d'un côté, la décomposition, le séquencement et le parallélisme des états, et d'un autre, la causalité, au niveau de la transition. Les relations entre états sont représentées à travers les tabulations qui expriment, dans un langage textuel, une décomposition. Les états séquentiels sont contenus dans un ConcernLevel et ont les mêmes coordonnées horizontales. Les états parallèles, cependant, sont contenus dans des ConcernLevel différents. Par ailleurs, la causalité des transitions est repérable grâce à la représentation linéaire des transitions, ainsi que les termes `When ... from ... to.`

8.4.7.La gestion de la complexité

La présentation actuelle du langage SCT est complexe. Plus la taille du système modélisé est importante, plus le modèle devient complexe au dépend de sa compréhension. Ceci est une conséquence directe de la représentation en masse de tous les aspects du système. Il devient ainsi difficile de repérer un concept particulier, de faire la part des modèles impliqués, ou même de mettre à jour le modèle sans le compromettre. C'est pourquoi il est nécessaire de considérer une séparation stricte et claire des modèles SCT_MV, SCT_MCA, SCT_MVA, SCT_MC, SCT_MP, et ce, dans des modules différents, afin de diviser cette complexité. Cette séparation a d'autant plus de sens, puisque le modèle SCT intervient dans des niveaux d'abstraction différents, tels que le domaine, l'application et l'adaptation.

8.4.8.L'intégration perceptive

La représentation de tous les concepts du langages SCT dans un modèle unique est certes complexe, mais elle facilite l'intégration d'un ensemble de concepts dans une perspective globale. Cependant, une solution qui assure cette intégration sans menacer la lisibilité du modèle serai de marier les deux, en mettant en place des vues séparées pour des points de vus différents et en maintenant une vue globale pour une compréhension macroscopique. Un menu de navigation listant l'ensemble des identifiants des variables,

des états et des niveaux de préoccupation facilite le repérage des concepts, mais des menus plus détaillés devront être mis en place pour permettre des actions telles que les dépendances avec d'autres concepts, ou l'orientation vers d'autres modèles. Ceci est d'autant plus important dans une perspective de séparation de modèles dans des vues disparates.

8.4.9. Le pouvoir d'expression visuel

Le pouvoir d'expression visuel est garanti par un ensemble de variables visuelles, qui facilitent le repérage des éléments clés du langage. En effet, une des particularités du Framework Xtext, utilisé dans le prototypage du langage, est sa capacité à mettre en valeur les terminaux de la grammaire du langage implémenté. De la sorte, les concepts clés du langage SCT, notamment, les types de variables, les types d'états, les niveaux de préoccupation, et les transitions sont mis en valeurs par un format particulier dans sa couleur et sa texture, i.e., les terminaux sont de couleur mauve et en texte gras. La position des éléments est aussi un facteur important dans leur expression visuelle. D'un côté, la position verticale facilite le repérage des concepts connexes (ex., les variables sont déclarées en premier, puis les états de la machine à états, et enfin, les transitions) et d'un autre, la position horizontale facilite la compréhension de la décomposition de la machine à état (ex., les états appartenant à un niveau de préoccupation son légèrement décalés à droite de celui-ci, de la même façon que les sous-états sont légèrement décalés à droite d'un état composite de plus haut niveau).

8.5. Conclusion

Deux types d'évaluations ont été effectuées pour déterminer le pouvoir d'expression, le passage à l'échelle et l'indépendance par rapport à un domaine d'un côté, et l'efficacité perceptive de la notation fournie d'un autre.

L'approche GQM a été suivie pour le premier volet de cette évaluation. Elle a permis d'établir que le langage SCT est expressif, car il est capable de spécifier la plupart des exigences définies (voir M1). Ensuite, vu que les systèmes comptant de 42 à 696 variables effectuent des opérations en quelques millisecondes (voir M2 et M3), il a été conclu que la représentation des concepts du langage SCT en tant que contraintes, dans un problème de satisfaction de contraintes, est une approche efficace et évolutive, car elle permet de

maintenir la performance requise, pour des systèmes d'échelles différentes. Enfin, le langage est capable d'être appliqué à différents domaines, y compris le domaine de l'Internet des Objet (IdO), les réseaux de capteurs sans fils (WSN) et les systèmes complexes (voir M4 et M5). Le Tableau 8-7 résume l'ensemble de ces résultats.

Question	Métrique	1er Cas	2ème Cas	3ème Cas
Q1	**M1**	17/19 = 89%		
Q2	**M2**	0.014363	0,006042	0.019769
	M3	1,3		
Q3	**M4**	3/3		
	M5	96%	100%	71%

Tableau 8-7 : Résumé des résultats de l'évaluation selon l'approche GQM

Bien que ces résultats soient satisfaisants, le problème principal reste lié aux exigences qui relèvent du temps. Qu'il s'agisse d'exigences de fréquence, d'ordre, ou même des exigences de sensibilité (AwRqs). Cette limitation provient de l'incapacité de la programmation par contraintes à supporter les divers aspects du temps. Certaines techniques ont été utilisées pour effectuer les tâches liées aux contraintes de temps, telles que l'exécution répétée du programme, similaire aux travaux de (Jackson 2002), ou l'introduction de variables de temps booléennes gérées par des processus externes. Par conséquent, bien qu'elles soient très courantes, les exigences qui s'activent à des heures précises de la journée, dans un ordre déterminé, par étape ou dont le ratio de succès doit être garanti dans une durée déterminée, sont en dehors des limites du langage, et nécessitent manifestement plus de recherche.

Dans un autre volet, les principes de la théorie perceptive, adaptés à une notation textuelle, ont permis de confirmer les facteurs qui facilitent la lecture et la compréhension d'un modèle selon la notation SCT. Cependant ils mettent en valeur les aspects qui nécessitent davantage d'améliorations. Le résumé des résultats de cette évaluation sont présentés dans le Tableau 8-8.

La couleur verte (++) détermine les propriétés assurées par la notation, la couleur orange (+) présente les propriétés partiellement prises en compte, mais qui peuvent profiter

d'améliorations, et finalement, le rouge (0), marque les propriétés qui limitent le pouvoir perceptif de la notation, et qui doivent être incorporés.

Principe	Propriété	Note	Commentaire
Clarté sémiotique	AR	++	Aucun concept n'est représenté doublement
	AS	+	Deux paires d'homonymes ont été détectées
	AE	++	Tout élément de la notation SCT à un méta-concept correspondant
	AD	++	Tous les concepts sont représentés par au moins un élément de la notation SCT
Distinction perceptive	DV	++	Verticale, pour grouper les concepts connexes, et horizontale, pour représenter la composition
	AP	0	Les éléments clés du langage SCT ne sont pas mis en valeur différemment
	CR	++	Les éléments clés de la syntaxe SCT sont mis en valeur par une texture et une couleur différente
Transparence sémantique	RP	+	Il est facile de repérer les méta-concepts relatifs aux éléments du langage, mis à part les événements et les contraintes qui ont la même syntaxe
	TS	++	Les relations entre les états sont perceptibles par les tabulations, et la causalité de la transition est perceptive par la linéarité de celle-ci, et le choix des terminaux de cet élément du langage
Gestion de la complexité	M	0	Le langage SCT ne permet pas la création de modèles séparés pour présenter des modules du langage différents
	A	0	Le langage SCT ne permet pas la création de modèles séparés pour présenter des niveaux d'abstraction différents du langage
Intégration cognitive	IC	+	L'intégration cognitive est assurée par un modèle unique. Mais une dualité entre globalité et partition peut être plus efficace
	IP	+	Un menu de navigation basique permet la détection des éléments à travers leurs identifiants. Il peut cependant offrir des fonctionnalités plus avancées de navigation et d'orientation
Pouvoir d'expression visuel	VV	+	Les variables visuelles incluent la couleur, la texture et la position verticale et horizontale des textes représentant les concepts clés du langage SCT

Tableau 8-8 : Résumé des résultats de l'évaluation selon la théorie perceptive

L'ensemble des limitations dévoilées à travers cette évaluation rentre dans les perspectives de cette thèse, et servent de base pour l'amélioration du langage.

Chapitre 9 Conclusions et perspectives

L'objectif principal de cette thèse est de spécifier les exigences dynamiques et évolutives des systèmes de systèmes dynamiques. La maîtrise de tels systèmes devient de plus en plus importante avec le développement et la croissance de paradigmes tels que l'internet des objets. En effet, les SdS_Dy sont aujourd'hui présents dans tous les secteurs de la vie de l'être humain ; dans l'industrie pour automatiser les processus de production, le domaine de la santé pour le suivi des malades, l'éducation, pour un e-learning façonné à chaque profil d'apprenant, l'agriculture, pour une prise en charge complète de tous les aspects reliés au maintien de plantes et ainsi de suite, pour une variété d'autres domaines. Son adoption par ces différents secteurs a été imposée par des besoins en rapidité, en performance, en disponibilité, en adaptabilité mais principalement, par les services innovants que ces systèmes permettent.

Les SdS_Dy sont des systèmes particuliers qui se caractérisent par leurs compositions dynamiques et évolutives, par leurs rôles indépendants mais aussi collaboratifs, et par leurs contextes internes (composition et configuration de l'ensemble des systèmes du SdS_Dy), externes (utilisateurs et environnement externes) et temporels variant. Une multitude de défis émergent cependant, d'abord dans l'identification de tous les éléments qui composent le SdS_Dy ou qui contribuent à son fonctionnement, ensuite dans la spécification de ces systèmes, et finalement, dans la prise en compte des évolutions dans leurs exigences et leurs compositions.

Ultimement, une meilleure compréhension du domaine des SdS_Dy s'est imposée. Pour ce faire, le cas détaillé d'un système d'irrigation est présenté. Il présente le portfolio GreenLife, ainsi que les divers cas d'utilisation souhaités par Maria, une cliente. Les évolutions d'exigences déclenchées par les changements dans le contexte de déploiement du SdS_Dy d'irrigation sont également discutés.

Le principal résultat de ce cas d'étude est **l'expression des exigences impliquées dans ce type de systèmes**, notamment, des exigences fonctionnelles, non fonctionnelles, structurelles, opérationnelles, paramétriques, de composition, de préférence, de proportionnalité, d'optimisation, de temps, de coût, d'autonomie, relaxables, d'optimisation, contextuelles, de sensibilité et d'évolution. La caractéristique commune distinguée pour toutes ces exigences est la variabilité, dans le temps et dans l'espace. Cette première contribution (Contr1) représente la réponse à la première question de recherche de cette thèse (**QR1**).

Sur la base de cette étude, deux principaux résultats ont été atteints :

- Évaluation des modèles de spécification de SdS_Dy. Dans ce sens, une variété d'approches, réputées pour leur capacité à spécifier les systèmes réactifs et dynamiques, ont été utilisé pour la spécification, et ont été analysé en conséquence. Dès lors, des modèles de trois paradigmes différents ont été réalisés, pour le cas du SdS_Dy GreenLife, du produit Maria : des modèles selon l'approche orienté par but, selon l'approche SysML, et finalement, selon l'approche des lignes de produits dynamiques (LdPD). Nous avons conclu que chacun de ces langages est efficace dans une catégorie d'exigences, ceci étant, aucun d'eux ne répond aux besoins spécifiques des SdS_Dy.
- Le choix de notre cadre de travail qui est le paradigme des lignes de produits logiciels dynamiques. Ce choix est motivé par la nature variable des exigences soulevées par l'étude de cas, et de l'importance de la gestion de cette variabilité en ingénierie des lignes de produits logiciels. Les LdPD correspondent particulièrement, car les SdS_Dy doivent s'adapter pendant leur exécution face à des contextes et des exigences dynamiques.

Ainsi, le **cadre "Software Product Lines for (4) Systems of Systems » (SPL4DSoS), a été introduit**, et correspond à la deuxième contribution de cette thèse (Contr2). Il est basé sur le Framework des LdPD et adapté aux caractéristiques spécifiques des SdS_Dy. Avec cela, nous répondons à la deuxième question de recherche (**QR2**).

Dans ce sens, la contribution principale de la thèse est SCT, un langage basé sur le cadre des LdPD adapté, qui capitalise sur les atouts de ce paradigme, en l'étendant avec des concepts de comportement dynamique et évolutif, dans le but d'enrichir son pouvoir

d'expression, afin de couvrir des systèmes tels que les SdS_Dy. Parmi les huit modèles composant le Framework SPL4DSoS, cinq sont implémentés dans le langage proposé, à savoir le modèle de variabilité de domaine, le modèle de variabilité d'application, le modèle de contexte, le modèle de configuration et un modèle de perception. Les modèles de connaissance, de contexte de domaine, et d'apprentissage ne sont pas pris en compte dans la version actuelle du langage SCT.

Dès lors, la troisième contribution (Contr3), à savoir **le langage State-Constraint Transition (SCT) pour la spécification formelle des exigences de SdS_Dy, est introduit.** D'une part, la syntaxe du langage est définie avec une grammaire EBNF, et d'une autre, la sémantique présente le sens de chacun des concepts composant le langage. Cette contribution présente une partie de la réponse à la troisième question de recherche (**QR3**).

Nous distinguons les états qui peuvent être états-contraintes simples ou états composites pour spécifier respectivement des exigences de bas et de haut niveaux. Les transitions qui déclenchent le passage du système d'un état de configuration à un autre. Les variables qui définissent les composants du système et de son contexte.

Et finalement, les contraintes qui spécifient les exigences imposées au niveau de chaque état.

De surcroît, et dans le but de remédier aux défis d'hétérogénéité et d'automatiser le processus de génération de configurations valides, tous les concepts du langage SCT sont **traduits en contraintes, générant ainsi un programme de contraintes.** Cette transformation correspond à la quatrième contribution (Contr4) et conclue la réponse à la troisième question de recherche (**QR3**).

Finalement, dans le but offrir aux ingénieurs la possibilité de spécifier les systèmes de systèmes dynamiques, et d'automatiser les tâches d'analyse et de reconfiguration, un outil devient utile.

Les deux aspects du langage sont ainsi prototypés dans le cadre du Framework Xtext. Ceci correspond à la cinquième et dernière contribution de cette thèse (Contr5), et répond à la quatrième question de recherche 4 (**QR4**).

Au bout du compte, cette thèse contribue dans le domaine des systèmes de systèmes en élaborant un langage qui assure la spécification formelle des exigences complexes, dynamiques et évolutives, et en automatisant la génération de configurations valides pour les SdS_Dy. Ceci est confirmé à travers une évaluation réalisée sur deux volets.

D'abord **l' efficacité du langage proposé à spécifier des SdS_Dy du monde réel est analysée** à travers l'évaluation de trois caractéristiques ; le pouvoir d'expression, le passage à l'échelle et d'indépendance au domaine. Ensuite, **l'efficacité cognitive du langage est examinée**, en se basant sur la théorie d'efficacité perceptive de Moody (Moody 2009), adaptée pour l'évaluation de langages textuels.

Les observations suivantes ont été constatées :

- Le langage SCT possède un pouvoir d'expression qui assure la représentation de 17/19 types d'exigences.
- La représentation sous forme de contraintes est une approche efficace et scalable, puisque le programme généré exécute des systèmes d'échelle très divergentes, en quelques millisecondes.
- Le langage ne dépend pas d'un domaine particulier, puisqu'il permet la spécification des exigences d'une application du domaine de l'IdO, de système WSN auto-adaptatifs et de systèmes complexes.
- La terminologie ainsi que les propriétés cognitives employées dans la langage SCT garantissent contribuent à la clarté sémiotique, à la distinction perceptive et à la transparence sémantique
- L'interface en une vue du modèle SCT compromet son intégration cognitive et son pouvoir d'expression visuel et amplifie en contrepartie sa complexité.

Les résultats réalisés dans le cadre de cette thèse contribuent implicitement dans le domaine des lignes de produits logiciels, en permettant ce qui suit :

- La définition de plusieurs niveaux de variabilité, à savoir la variabilité de domaine, qui présente toutes les caractéristiques d'un systèmes dans un domaine donné, pour la réutilisation lors de la création de produits, ainsi que la variabilité d'application, qui présente, pour une dérivation spécifique de la ligne de produits, les différentes possibilités de configuration, ainsi que les différents modes de fonctionnement, pour la réutilisation lors de l'adaptation du produit dans un contexte spécifique.

- La définition des évolutions des modèles de variabilité, conséquence de l'évolution des exigences, et ce en considérant un modèle source flexible, ainsi qu'un modèle de variabilité d'application qui explore les nouvelles exigences imposées selon le contexte.
- La définition de machines à états finis, pour définir un ensemble d'états de configurations possibles pour chaque dérivation du SdS_Dy. Chaque état définit un ensemble de contraintes, représentatives des exigences imposées dans un contexte donné. La satisfaction de ces contraintes donne lieu à une ou plusieurs configurations valides. Les transitions d'un état de configuration vers un autre, sont représentatives d'un événement du contexte, qui exige ou impose la recherche d'une nouvelle configuration valide, ou plus performante.
- La définition d'éléments multi-instanciés, représentatifs des systèmes de même nature, qui sont présents dans le SdS_Dy en plusieurs occurrences. Ces systèmes, existant réellement dans des états différents, sont susceptibles de subir des contraintes différentes. La gestion des multi-instances est ainsi prise en compte dans le modèle SCT.

Cependant, bien que les problèmes de recherche motivant cette thèse soient traités à travers ces contributions, certaines améliorations ont été identifiées.

- La **première perspective** est liée à l'implémentation du langage SCT afin de permettre la configuration dynamique effective des systèmes. Dans ce sens, les composants de chaque opérationnalisation sont implémentés, ainsi que leurs dépendances, et le processus de configuration, réparti ou centralisé est établi.
- La **deuxième perspective** est liée à la vérification de la cohérence dans le problème de contraintes généré. En effet, lorsque le problème de contraintes est livré au solveur, celui-ci doit vérifier que toutes les contraintes sont satisfaites afin de proposer une (des) valeur (s) pour chaque variable. Cependant, lorsque certaines parmi les contraintes exprimées ou générées sont contradictoires, le solveur n'est plus capable de générer une solution valide. Dans cette thèse, la validation est réalisée de manière manuelle, en parcourant les contraintes, et en détectant les sources de conflit. L'automatisation de cette opération, étant primordial pour tout problème de contraintes, n'est pas adressée, et doit par conséquent être considérée dans les travaux en perspective.

- La **troisième perspective** est liée à tout ce qui relève du temps ; notamment, à l'ordonnancement de l'exécution des contraintes, à la spécification d'une contrainte liée au temps réel, à la planification ou la répétition de résolution d'un problème de contraintes. Ceci correspond, dans le cadre du langage SCT, à un état qui s'active dans un temps précis ou avec une fréquence définie, ou à un ensemble d'états qui s'exécutent selon un ordre particulier. Ce problème se pose également lorsqu'une série de reconfigurations intermédiaires est nécessaire pour parvenir à une reconfiguration plus importante. Cette question a en effet été soulevée pour les SdS_Dy, car diverses modifications dans des petites parties du SdS sont parfois nécessaires pour atteindre une nouvelle configuration plus significative.
- La **quatrième perspective** relève de l'intégration de tous les modèles proposés dans le SPL4DSoS, notamment le modèle d'environnement, le modèle de contexte de domaine, et le modèle d'apprentissage, ainsi que les dépendances qu'ils ont avec les autres modèles du Framework. En effet, toutes les connaissances considérées comme entrées dans un modèle SCT sont fournies par des ingénieurs et des experts du domaine. Cependant, pour faire face à l'évolution technologique rapide actuelle, en particulier à l'ère de l'intelligence artificielle, de l'apprentissage automatique et des marchés publics (Marketplace), tout changement, toute nouvelle possibilité ou variation du domaine doit être automatiquement détecté par les modèles d'apprentissage, et injectés dans les modèles de domaine.
- La **cinquième perspective** vise à mettre à niveau l'efficacité perceptive du modèle, en incorporant les conclusions du chapitre sur l'évaluation. Principalement, en proposant des vues séparées pour les différents points de vue d'un modèle selon la notation SCT et en proposant des combinaisons de variables visuelles qui mettent en valeur des concepts différents distinctement.
- La **sixième et dernière perspective** vise à formaliser la sémantique du langage SCT qui est décrite dans cette thèse, en langage naturel. La formalisation est une étape nécessaire dans la définition d'un tel langage, afin d'unifier sa compréhension, et faciliter ses éventuelles évolutions et extensions.

Références

Abbas, Nadeem, Jesper Andersson, and Welf Löwe. 2010. “Autonomic Software Product Lines (ASPL).” *ACM International Conference Proceeding Series*, no. January: 324–31. https://doi.org/10.1145/1842752.1842812.

Abowd, Gregory D., Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. 1999. “Towards a Better Understanding of Context and Context-Awareness.” In *Nternational Symposium on Handheld and Ubiquitous Computing*, 304–7. Berlin, Heidelberg: Springer. https://doi.org/10.1007/3-540-48157-5_29.

Achtaich, Asmaa. 2016. “Vers Une Plateforme de Monitoring de Flottes d’objets Connectés.” In *Actes Du 8 e Forum Jeunes Chercheurs Du Congrès INFORSID*, 9. Grenoble.

Achtaich, Asmaa, Raúl Mazo, Nissrine Souissi, Camille Salinesi, and Ounsa Roudies. 2019. “Management Capabilities for Mobile and IoT Devices: An Evaluation Framework.” *International Journal of Engineering and Advanced Technology (IJEAT)*, no. 6: 2249–8958. https://doi.org/10.35940.

Achtaich, Asmaa, Ounsa Roudies, Nissrine Souissi, and Camille Salinesi. 2015. “Selecting SPL Modeling Languages: A Practical Guide.” In *Proceedings of 2015 IEEE World Conference on Complex Systems, WCCS 2015*, 1–6. IEEE. https://doi.org/10.1109/ICoCS.2015.7483312.

Achtaich, Asmaa, Ounsa Roudies, Nissrine Souissi, Camille Salinesi, and Raúl Mazo. 2019. “Evaluation of the State-Constraint Transition Modeling Language: A Goal Question Metric Approach.” In *Software Product Line Conference Proceedings - Volume B*. Paris. https://doi.org/https://doi.org/10.1145/3307630.3342417.

Achtaich, Asmaa, Nissrine Souissi, Raul Mazo, Ounsa Roudies, and Camille Salinesi. 2018. “A DSPL Design Framework for SASs: A Smart Building Example.” *EAI Endorsed Transactions on Smart Cities* 2 (8): 154829. https://doi.org/10.4108/eai.26-6-2018.154829.

Achtaich, Asmaa, Nissrine Souissi, Raul Mazo, Camille Salinesi, and Ounsa Roudies. 2017. “Designing a Framework for Smart IoT Adaptations.” In *International Conference on Emerging Technologies for Developing Countries*, edited by Springer. Marrakech-Morocco. https://www.researchgate.net/profile/Asmaa_Achtaich/publication/316544423_Designing_a_Framework_for_Smart_IoT_Adaptations/links/5925d476aca27295a8f1368c/Designing-a-Framework-for-Smart-IoT-Adaptations.pdf.

Achtaich, Asmaa, Nissrine Souissi, Ounsa Roudies, Camille Salinesi, and Raúl Mazo. 2019. “A Constraint-Based Approach to Deal with Self-Adaptation: The Case of Smart Irrigation Systems.” *International Journal of Advanced Computer Science and Applications* 10 (7): 184–93. https://doi.org/10.14569/ijacsa.2019.0100727.

Ahmad, Manzoor, Jean-Michel Bruel, Regine Laleau, and Christophe Gnaho. 2012. “Using RELAX, SysML and KAOS for Ambient Systems Requirements Modeling.” *Procedia Computer Science* 19: 474 – 481.

Alawairdhi, Mohammed, and Eisa Aleisa. 2011. “A Scenario Based Approach for Requirements Elicitation for Software Systems Complying with the Utilization of Ubiquitous Computing Technologies.” In

Proceedings - International Computer Software and Applications Conference, 341–44. https://doi.org/10.1109/COMPSACW.2011.63.

Alférez, Germán H., Vicente Pelechano, Raúl Mazo, Camille Salinesi, and Daniel Diaz. 2014. "Dynamic Adaptation of Service Compositions with Variability Models." *Journal of Systems and Software* 91 (1): 24–47. https://doi.org/10.1016/j.jss.2013.06.034.

Almaliki, Malik, Funmilade Faniyi, Rami Bahsoon, Keith Phalp, and Raian Ali. 2014. "Requirements-Driven Social Adaptation: Expert Survey." *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8396 LNCS: 72–87. https://doi.org/10.1007/978-3-319-05843-6_6.

Anda, Bente, Kai Hansen, Ingolf Gullesen, and Hanne Kristin Thorsen. 2006. "Experiences from Introducing UML-Based Development in a Large Safety-Critical Project." *Empirical Software Engineering* 11 (4): 555–81. https://doi.org/10.1007/s10664-006-9020-6.

Antonino, Pablo Oliveira, Andreas Morgenstern, Benno Kallweit, Martin Becker, and Thomas Kuhn. 2018. "Straightforward Specification of Adaptation-Architecture-Significant Requirements of IoT-Enabled Cyber-Physical Systems." *Proceedings - 2018 IEEE 15th International Conference on Software Architecture Companion, ICSA-C 2018*, 19–26. https://doi.org/10.1109/ICSA-C.2018.00012.

Asikainen, Timo, Tomi Männistö, and Timo Soininen. 2004. "Using a Configurator for Modelling and Configuring Software Product Lines Based on Feature Models." In *Workshop On Software Variability Management For Product Derivation, Software Product Line Conference (SPLC3)*, 24--35.

Baccelli, Emmanuel, Cenk Gundogan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wahlisch. 2018. "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT." *IEEE Internet of Things Journal* 5 (6): 4428–40. https://doi.org/10.1109/JIOT.2018.2815038.

Bachmann, Felix, and Paul C Clements. 2005. "Variability in Software Product Lines."

Batory, Don. 2005. "Feature Models, Grammars, and Propositional Formulas." In *Lecture Notes in Computer Science*, 7–20. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11554844_3.

Baumer, Eric P.S., Vera Khovanskaya, Mark Matthews, Lindsay Reynolds, Victoria Schwanda Sosik, and Geri Gay. 2014. "Reviewing Reflection: On the Use of Reflection in Interactive System Design." *Proceedings of the 2014 Conference on Designing Interactive Systems - DIS '14*, 93–102. https://doi.org/10.1145/2598510.2598598.

Becker, Matthias, Markus Luckey, and Steffen Becker. 2013. "Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time." *QoSA 2013 - Proceedings of the 9th International ACM Sigsoft Conference on the Quality of Software Architectures*, no. May 2014: 43–52. https://doi.org/10.1145/2465478.2465489.

Benavides, David, Sergio Segura, and Antonio Ruiz-cort. 2010. "Automated Analysis of Feature Models 20 Years Later : A Literature Review 6." *Information Systems* 35 (6): 615–36. https://doi.org/10.1016/j.is.2010.01.001.

Benavides, David, Pablo Trinidad, and Antonio Ruiz Cortés. 2005. "Using Constraint Programming to Reason on Feature Models." *Seke*, 677–82.

Bencomo, Nelly, Peter Sawyer, Gordon S. Blair, and Paul Grace. 2008. "Dynamically Adaptive Systems Are

Product Lines Too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems." *Splc (2)*, 23–32.

Benlarabi, Anissa. 2014. "Towards a Co-Evolution Model for Software Product Lines Based on Cladistics." In *2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS)*, 1–6. IEEE. https://doi.org/10.1109/RCIS.2014.6861084.

Bettini, Lorenzo. 2013. *Implementing Domain-Specific Languages with Xtext and Xtend*. *Packt Publishing*. Packt Publishing.

Bocanegra, Jose, Jaime Pavlich-Mariscal, and Angela Carrillo-Ramos. 2016. "On the Role of Model-Driven Engineering in Adaptive Systems." *2016 IEEE 11th Colombian Computing Conference, CCC 2016 - Conference Proceedings*. https://doi.org/10.1109/ColumbianCC.2016.7750792.

Boniol, Frédéric, and Virginie Wiels. 2014. "Landing Gear System."

Bresciani, Paolo, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. 2004. "Tropos: An Agent-Oriented Software Development Methodology." *Autonomous Agents and Multi-Agent Systems* 8 (3): 203–36. https://doi.org/10.1023/B:AGNT.0000018806.20944.ef.

Bürdek, Johannes, Sascha Lity, Malte Lochau, Markus Berens, Ursula Goltz, and Andy Schürr. 2013. "Staged Configuration of Dynamic Software Product Lines with Complex Binding Time Constraints." *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems - VaMoS '14*, 1–8. https://doi.org/10.1145/2556624.2556627.

Capilla, Rafael, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. 2014. "An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry." *Journal of Systems and Software* 91 (1): 3–23. https://doi.org/10.1016/j.jss.2013.12.038.

Cetina, Carlos, Øystein Haugen, Xiaorui Zhang, Franck Fleurey, and Vicente Pelechano. 2009. "Strategies for Variability Transformation at Run-Time." *Proceedings of the 13th International Software Product Line Conference*, no. November 2016: 61–70. https://doi.org/10.1145/1753235.1753245.

Chakraborty, Abhijit, Mrinal Kanti Baowaly, Ashraful Arefin, and Ali Newaz Bahar. 2012. "The Role of Requirement Engineering in Software Development Life Cycle." *Journal of Emerging Trends in Computing and Information Sciences* 3 (5): 723–29.

Chinrungrueng, Jatuporn, Udomporn Sunantachaikul, and Satien Triamlumlerd. 2007. "Smart Parking: An Application of Optical Wireless Sensor Network." In *International Symposium on Applications and the Internet Workshops (SAINTW'07)*. IEEE.

Chun, Ingeol, Jeongmin Park, Haeyoung Lee, Wontae Kim, Seungmin Park, and Eunseok Lee. 2013. "An Agent-Based Self-Adaptation Architecture for Implementing Smart Devices in Smart Space." *Telecommunication Systems* 52 (4): 2335–46. https://doi.org/10.1007/s11235-011-9547-8.

Classen, Andreas, Quentin Boucher, and Patrick Heymans. 2011. "A Text-Based Approach to Feature Modelling: Syntax and Semantics of TVL." *Science of Computer Programming* 76 (12): 1130–43. https://doi.org/10.1016/j.scico.2010.10.005.

Clauß, Matthias, and Intershop Jena. 2001. "Modeling Variability with UML." *In GCSE 2001 Young Researchers Workshop.* https://doi.org/10.1.1.109.3401.

Clements, Paul, and Linda. Northrop. 2002. *Software Product Lines : Practices and Patterns*. Addison-Wesley.

Cordy, Maxime, Pierre-yves Schobbens, Patrick Heymans, and Axel Legay. 2013. "Beyond Boolean Product-

Line Model Checking : Dealing with Feature Attributes and Multi-Features." In *35th International Conference on Software Engineering (ICSE)*, 472–81. IEEE.

Czarnecki, Krzysztof. 1998. "Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models."

Czarnecki, Krzysztof, and Chang Hwan Peter Kim. 2005. "Cardinality-Based Feature Modeling and Constraints: A Progress Report." *International Workshop on Software Factories*, 16–20. https://doi.org/1-59593-193-7/05/0010.

D'Ippolito, Nicolas, Víctor Braberman, Jeff Kramer, Jeff Magee, Daniel Sykes, and Sebastian Uchitel. 2014. "Hope for the Best, Prepare for the Worst: Multi-Tier Control for Adaptive Systems." *Proceedings of the 36th International Conference on Software Engineering*. https://doi.org/10.1145/2568225.2568264.

Danny Weyns. 2017. "Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges." *Handbook of Software Engineering*, 1–41.

Dar, Hafsa, M. Ikramullah Lali, Humaira Ashraf, Muhammad Ramzan, Tehmina Amjad, and Basit Shahzad. 2018. "A Systematic Study on Software Requirements Elicitation Techniques and Its Challenges in Mobile Application Development." *IEEE Access* 6: 63859–67. https://doi.org/10.1109/ACCESS.2018.2874981.

Dardenne, Anne, Axel van Lamsweerde, and Stephen Fickas. 1993. "Goal-Directed Requirements Acquisition." *Science of Computer Programming* 20 (1–2): 3–50. https://doi.org/10.1016/0167-6423(93)90021-G.

Darwish, Nagy Ramadan, and Bassem S M Zohdy. 2016. "Goal Modeling Techniques for Requirements Engineering Software Engineering." *International Journal of Computer Science and Information Security* 14 (7).

Dechter, Rina, and Judea Pearl. 1988. "Network-Based Heuristics for Constraint-Satisfaction Problems." In *Search in Artificial Intelligence*, 370–425. New York, NY: Springer New York. https://doi.org/10.1007/978-1-4613-8788-6_11.

Delligatti, Lenny. 2014. *SysML Distilled :A Brief Guide to the Systems Modeling Language*. Addison Wesley.

Denko, Mieso K., Laurence Tianruo Yang, and Yan Zhang. 2009. "Software Architecture-Based Self-Adaptation." *Autonomic Computing and Networking*, 1–458. https://doi.org/10.1007/978-0-387-89828-5.

Dhungana, Deepak, Paul Grünbacher, and Rick Rabiser. 2011. "The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: A Multiple Case Study." *Automated Software Engineering* 18 (1): 77–114. https://doi.org/10.1007/s10515-010-0076-6.

Djebbi, Olfa, and Camille Salinesi. 2006. "Criteria for Comparing Requirements Variability Modeling Notations for Product Lines." In *Fourth International Workshop on Comparative Evaluation in Requirements Engineering (CERE'06 - RE'06 Workshop)*, 20–35. IEEE. https://doi.org/10.1109/CERE.2006.2.

Djebbi, Olfa, Camille Salinesi, and Daniel Diaz. 2007. "Deriving Product Line Requirements: The RED-PL Guidance Approach." In *14th Asia-Pacific Software Engineering Conference (APSEC'07)*, 494–501. IEEE. https://doi.org/10.1109/ASPEC.2007.63.

Dounas, Lamiae, Raúl Mazo, Camille Salinesi, and Omar El-Beqqali. 2015. "Continuous Monitoring of

Adaptive E-Learning Systems Requirements." In *12th International Conference on Computer Systems and Applications*, 17–20. Marrakech.

Duarte, Bruno Borlini, Andre Luiz De Castro Leal, Ricardo De Almeida Falbo, Giancarlo Guizzardi, Renata S.S. Guizzardi, and Silva Vítor E. Souza. 2018. "Ontological Foundations for Software Requirements with a Focus on Requirements at Runtime." *Applied Ontology* 13 (2): 73–105. https://doi.org/10.3233/AO-180197.

Duarte, Bruno Borlini, Vítor E.Silva Souza, André Luiz De Castro Leal, Ricardo De Almeida Falbo, Giancarlo Guizzardi, and Renata S.S. Guizzardi. 2016. "Towards an Ontology of Requirements at Runtime." *Frontiers in Artificial Intelligence and Applications* 283: 255–68. https://doi.org/10.3233/978-1-61499-660-6-255.

Dumitrescu, Cosmin. 2014. "CO-OVM: A Practical Approach to Systems Engineering Variability Modeling." Université Paris 1 Panthéon Sorbonne.

Dumitrescu, Cosmin, Raul Mazo, Camille Salinesi, and Alain Dauron. 2013. "Bridging the Gap between Product Lines and Systems Engineering." In *Proceedings of the 17th International Software Product Line Conference on - SPLC '13*, 254. New York, New York, USA: ACM Press. https://doi.org/10.1145/2491627.2491655.

Elsayed, Iman Ahmed, Zeinab Ezz, and Eman Nasr. 2017. "Goal Modeling Techniques in Requirements Engineering: A Systematic Literature Review." *Journal of Computer Science*. https://doi.org/10.3844/jcssp.2017.430.439.

Elsner, Christoph, Goetz Botterweck, Daniel Lohmann, and Wolfgang Schröder-Preikschat. 2010. "Variability in Time - Product Line Variability and Evolution Revisited." In *VaMoS*.

Elsner, Von, D. Briassoulis, D. Briassoulis, A. Waaijenberg, Chr Von Zabeltitz Mistriotis, J. Gratraud, G. Russo, and R. Suay-Cortes. 2000. "Review of Structural and Functional Characteristics of Greenhouses in European Union Countries: Part I, Design Requirements." *Journal of Agricultural Engineering Research* 75 (1): 1–16. https://doi.org/10.1006/jaer.1999.0502.

Eriksson, Magnus, Jürgen Börstler, and Kjell Borg. 2005. "The PLUSS Approach – Domain Modeling with Features, Use Cases and Use Case Realizations." In , 33–44. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11554844_5.

Estublier, Jacky. 2000. "Software Configuration Management A Road Map." In *ICSE-Future Of SE Trac*.

Favaro, John, and Silvia Mazzini. 2009. "Extending FeatuRSEB with Concepts from Systems Engineering." In , 41–50. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-04211-9_5.

Fernandes, Paula, Cláudia Werner, and Leonardo Gresta Paulino Murta. 2008. "Feature Modeling for Context-Aware Software Product Lines." In *SEKE*.

Feynman, Richard, and Chapter Objectives. 2016. *EBNF A Notation to Describe Syntax*.

Filipe, Joaquim, Ana Fred, and Bernadette Sharp. 2011. "Toward a Self-Adaptive Multi-Agent System to Control Dynamic Processes." *Communications in Computer and Information Science* 129. https://doi.org/10.1007/978-3-642-19890-8.

Fredericks, Erik M., Byron Devries, and Betty H.C. Cheng. 2014. "AutoRELAX: Automatically RELAXing a Goal Model to Address Uncertainty." *Empirical Software Engineering* 19 (5): 1466–1501. https://doi.org/10.1007/s10664-014-9305-0.

Freuder, Eugene C., and Richard J. Wallace. 1992. “Partial Constraint Satisfaction.” *Artificial Intelligence* 58 (1–3): 21–70. https://doi.org/10.1016/0004-3702(92)90004-H.

Gorod, A., B. Sauser, and J. Boardman. 2008. “System-of-Systems Engineering Management: A Review of Modern History and a Path Forward.” *Systems Journal* 2 (4): 484–99. https://doi.org/10.1109/JSYST.2008.2007163.

Granjal, Jorge, Edmundo Monteiro, and Jorge Sá Silva. 2014. “Security in the Integration of Low-Power Wireless Sensor Networks with the Internet: A Survey.” *Ad Hoc Networks* 24 (Part A): 264–87. https://doi.org/10.1016/j.adhoc.2014.08.001.

Gurp, Jilles Van, Jan Bosch, and Mikael Svahnberg. 2001. “On the Notion of Variability in Software Product Lines By On the Notion of Variability in Software Product Lines.” In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, 45. IEEE Computer Society.

Hallsteinsen, S., M. Hinchey, Sooyong Park, and K. Schmid. 2008. “Dynamic Software Product Lines.” *Computer* 41 (4): 93–95. https://doi.org/10.1109/MC.2008.123.

Han, Deshuai, Jianchun Xing, Qiliang Yang, Juelong Li, Xiaobing Zhang, and Ying Chen. 2017. “Integrating Goal Models and Problem Frames for Requirements Analysis of Self-Adaptive CPS.” *Proceedings - International Computer Software and Applications Conference* 2: 529–35. https://doi.org/10.1109/COMPSAC.2017.152.

Harel, David. 1987. “On the Formal Semantics of Statecharts.Pdf.” In *2nd IEEE Symposium on Logic in Computer Science*.

Harel, David, and Bernhard Rumpe. 2000. “Modeling Languages: Syntax, Semantics and All That Stuff.” https://doi.org/10.1.1.10.1512.

Harper, Richard. 2003. *Inside the Smart Home*. Springer S. Springer.

Hein, Andreas M., Yann Chazal, Samuel Boutin, and Marija Jankovic. 2018. “A Methodology for Architecting Collaborative Product Service System of Systems.” In *2018 13th System of Systems Engineering Conference, SoSE 2018*, 53–59. Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/SYSOSE.2018.8428697.

Hentenryck, Fascal Van. 1987. “A Theoretical Framework for Consistency Techniques in Logic Programming.” In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, 2–8. MIT Press.

Hevner, Alan R., Salvatore T. March, Jinsoo Park, and Sudha Ram. 2004. “Design Science in Information Systems Research.” *MIS Quarterly* 28 (1): 75–105.

Hevner, Alan R. 2007. “A Three Cycle View of Design Science Research.” *Scandinavian Journal of Information Systems* 19 (2): 87–92.

Hooman, J.J.M., S. Ramesh, W.P. De Roever, J.W. Klop, J.J.C. Meyer, and J.J.M.M. Rutten. 1988. “A Compositional Semantics for Statecharts.” In *Eurographics ’88 Liber Amicorum: JW de Bakker, 25 Jaar Semantiek*, 275–87.

Hotz, Lothar, Andreas Günter, and Thorsten Krebs. 2003. “A Knowledge-Based Product Derivation Process and Some Ideas How to Integrate Product Development.” In *Software Variability Management*.

Hotz, Lothar, Stephanie Von Riegen, Rainer Herzog, Matthias Riebisch, and Markus Kiele-Dunsche. 2019. “Adaptive Autonomous Machines-Requirements and Challenges.” In *21th International Configuration*

Workshop, 61.

Hughes, Danny, Phil Greenwood, Geoff Coulson, and Gordon Blair. 2006. "GridStix: Supporting Flood Prediction Using Embedded Hardware and Next Generation Grid Middleware." In *International Symposium on a World of Wireless, Mobile and Multimedia Networks(WoWMoM'06)*, 621–26. IEEE. https://doi.org/10.1109/WOWMOM.2006.49.

IBM. 2005. "Autonomic Computing White Paper: An Architectural Blueprint for Autonomic Computing." *IBM White Paper*, no. June: 34. https://doi.org/10.1021/am900608j.

IEEE. 2009. "830-1998 - IEEE Recommended Practice for Software Requirements Specifications." https://standards.ieee.org/standard/830-1998.html.

INCOSE. 2018. "Systems Engineering Handbook." *INSIGHT* 1 (2): 20–20. https://doi.org/10.1002/inst.19981220.

Ingolfo, Silvia, and Silva Vitor E. Souza. 2013. "Law and Adaptivity in Requirements Engineering." *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, no. May 2013: 163–68. https://doi.org/10.1109/SEAMS.2013.6595503.

ISO. 2017. "ISO - ISO/IEC/IEEE 12207:2017 - Systems and Software Engineering — Software Life Cycle Processes." https://www.iso.org/standard/63712.html.

ITU-T, & Intern. Telecom. Union: Recommendation Z.151. 2008. "Z.151 : User Requirements Notation (URN) - Language Definition."

Jackson, Daniel. 2002. "Alloy: A Lightweight Object Modelling Notation." *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11 (2): 256–290.

Jaroucheh, Zakwan, Xiaodong Liu, and Sally Smith. 2010. "CANDEL: Product Line Based Dynamic Context Management for Pervasive Applications." In *The Fourth International Conference on Complex, Intelligent and Software Intensive Systems*, 209–16. Poland.

Jazayeri, Mehdi, Alexander Ran, and Frank van der Linden. 2000. *Software Architecture for Product Families: Principles and Practice*.

Kamienski, Carlos, Juha-Pekka Soininen, Markus Taumberger, Ramide Dantas, Attilio Toscano, Tullio Salmon Cinotti, Rodrigo Filev Maia, and André Torre Neto. 2019. "Smart Water Management Platform: IoT-Based Precision Irrigation for Agriculture." *Sensors (Basel, Switzerland)* 19 (2): 276. https://doi.org/10.3390/s19020276.

Kang, Kyo C, Sholom G Cohen, James a Hess, William E Novak, and a Spencer Peterson. 1990. "Feature-Oriented Domain Analysis (FODA) Feasibility Study." *Distribution* 17 (November): 161. https://doi.org/10.1080/10629360701306050.

Klein, Heinz K., and Michael D. Myers. 1999. "A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems." *MIS Quarterly* 23 (1): 67. https://doi.org/10.2307/249410.

Krebs, Thorsten, Lothar Hotz, and Andreas Günter. 2002. "Knowledge-Based Configuration for Configuring Combined Hardware/Software Systems." In *IN PROC. OF 16. WORKSHOP, PLANEN, SCHEDULING UND KONFIGURIEREN, ENTWERFEN*, 10--11.

Krupitzer, Christian, Martin Pfannemüller, Vincent Voss, and Christian Becker. 2018. "Comparison of Approaches for Developing Self-Adaptive Systems."

Kumar, Manish, Nirav Ajmeri, and Smita Ghaisas. 2010. "Towards Knowledge Assisted Agile Requirements

Evolution." In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering - RSSE '10*, 16–20. New York, New York, USA: ACM Press. https://doi.org/10.1145/1808920.1808924.

L'Obs. 2016. "La Chaudière Connectée Appelle Toute Seule La Maintenance : 134.000 Euros de Facture - 17 Juin 2016 - L'Obs." 2016. http://tempsreel.nouvelobs.com/l-histoire-du-soir/20160617.OBS2894/la-chaudiere-connectee-appelle-toute-seule-la-maintenance-134-000-euros-de-facture.html.

Lamsweerde, A. van. 2001. "Goal-Oriented Requirements Engineering: A Guided Tour." In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, 249–62. IEEE Comput. Soc. https://doi.org/10.1109/ISRE.2001.948567.

Lauenroth, Kim, and Klaus Pohl. 2007. "Towards Automated Consistency Checks of Product Line Requirements Specifications." *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering - ASE '07*, 373. https://doi.org/10.1145/1321631.1321687.

Lee, Euijong, Young Gab Kim, Young Duk Seo, Kwangsoo Seol, and Doo Kwon Baik. 2018. "RINGA: Design and Verification of Finite State Machine for Self-Adaptive Software at Runtime." *Information and Software Technology* 93 (January): 200–222. https://doi.org/10.1016/j.infsof.2017.09.008.

Lee, Seung Min, Soojin Park, and Young B. Park. 2019. "Self-Adaptive System Verification Based on SysML." In *ICEIC 2019 - International Conference on Electronics, Information, and Communication*. Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.23919/ELINFOCOM.2019.8706383.

Lin, Jinxin, Mark S Fox, and Taner Bilgic. 1996. "A Requirement Ontology for Engineering Design." *Concurrent Engineering* 4 (3): 279–91. https://doi.org/10.1177/1063293X9600400307.

Macías-Escrivá, Frank D., Rodolfo Haber, Raul Del Toro, and Vicente Hernandez. 2013. "Self-Adaptive Systems: A Survey of Current Approaches, Research Challenges and Applications." *Expert Systems with Applications* 40 (18): 7267–79. https://doi.org/10.1016/j.eswa.2013.07.033.

Mahya, Parisa, and Hooman Tahayori. 2016. "IoT Is SoS." In *Int'l Conf. Internet Computing and Internet of Things*, 38–42.

Mannion, Mike, Juha Savolainen, and Timo Asikainen. 2009. "Viewpoint-Oriented Variability Modeling." *Proceedings - International Computer Software and Applications Conference* 1: 67–72. https://doi.org/10.1109/COMPSAC.2009.19.

Mavin, Alistair, Philip Wilkinson, Adrian Harwood, and Mark Novak. 2009. "EARS (Easy Approach to Requirements Syntax)." In *Proceedings of the IEEE International Conference on Requirements Engineering*, 317–22. https://doi.org/10.1109/RE.2009.9.

Mazo, Raul. 2018. *Guía Para La Adopción Industrial de Líneas de Productos de Software*. Universidad EAFIT. https://www.amazon.in/adopción-industrial-productos-software-Spanish-ebook/dp/B07G83K23C.

Mazo, Raúl. 2014. "Avantages et Limites Des Modèles de Caractéristiques Dans La Modélisation Des Exigences de Variabilité." *Revue Génie Logiciel* 111: 42–48.

———. 2018. "Software Product Lines, from Reuse to Self Adaptive Systems." Université Paris 1 Panthéon - Sorbonne, France.

Mazo, Raúl, Cosmin Dumitrescu, Camille Salinesi, and Daniel Diaz. 2014. "Recommendation Heuristics for Improving Product Line Configuration Processes." *Recommendation Systems in Software Engineering*, 511–37. https://doi.org/10.1007/978-3-642-45135-5_19.

Mazo, Raúl, Carlos Jaramillo, Paola Vallejo, and Jhon Medina. 2020. "Towards a New Template for the Specification of Requirements in Semi-Structured Natural Language." *Journal of Software Engineering Research and Development* 6 (1).

Mazo, Raúl, Camille Salinesi, Daniel Diaz, Juan C Muñoz-Fernández, Luisa Rincón, Camille Salinesi, and Gabriel Tamura. 2015. "VariaMos: An Extensible Tool for Engineering (Dynamic) Product Lines." In *Proceedings of the 24th International Conference on Advanced Information Systems Engineering (CAiSE Forum'12)*, 374–79. https://doi.org/10.1145/2791060.2791103.

Mazo, Raúl, Camille Salinesi, Olfa Djebbi, Daniel Diaz, and Alberto Lora-Michiels. 2012. "Constraints: The Heart of Domain and Application Engineering in the Product Lines Engineering Strategy." *International Journal of Information System Modeling and Design* 3 (2).

Mecibah, Zina, and Fateh Boutekkouk. 2019. *Towards Requirements Engineering Process for Self-Adaptive Embedded Systems*. *Csoc 2018, Aisc 763*. Vol. 763. Springer International Publishing. https://doi.org/10.1007/978-3-319-91186-1.

Metzger, Andreas, and Klaus Pohl. 2014. "Software Product Line Engineering and Variability Management: Achievements and Challenges." *Proceedings of the on Future of Software Engineering*, no. JUNE: 70–84. https://doi.org/10.1145/2593882.2593888.

Minizinc. 2019. "MiniZinc." 2019. https://www.minizinc.org/.

Moody, Daniel. 2009. "The Physics of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering." *IEEE Transactions on Software Engineering* 35 (6): 756–79. https://doi.org/10.1109/TSE.2009.67.

Moreno, M. Victoria, Miguel A. Zamora, and Antonio F. Skarmeta. 2015. "An IoT Based Framework for User-Centric Smart Building Services." *International Journal of Web and Grid Services* 11 (1): 78. https://doi.org/10.1504/IJWGS.2015.067157.

Mori, Marco. 2012. "A Framework to Support Consistent Design and Evolution of Adaptive Systems." IMT Institute for Advanced Studies, Lucca.

Morin, Brice, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. 2009. "Models@ Run.Time to Support Dynamic Adaptation." *Computer* 42 (10): 44–51. https://doi.org/10.1109/MC.2009.327.

Morin, Brice, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair. 2008. "An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability." In *Model Driven Engineering Languages and Systems*, 782–96. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-87875-9_54.

Muñoz-Fernández, Juan C., Alessia Knauss, Lorena Castañeda, Mahdi Derakhshanmanesh, Robert Heinrich, Matthias Becker, and Nina Taherimakhsousi. 2017. "Capturing Ambiguity in Artifacts to Support Requirements Engineering for Self-Adaptive Systems." *CEUR Workshop Proceedings* 1796.

Muñoz-Fernández, Juan C., Raúl Mazo, Camille Salinesi, and Gabriel Tamura. 2018. "10 Challenges for the Specification of Self-Adaptive Software." *Proceedings - International Conference on Research Challenges in Information Science* 2018–May (June 2019): 1–12. https://doi.org/10.1109/RCIS.2018.8406640.

Muñoz-Fernández, Juan C., Gabriel Tamura, Mazo Raúl, and Camille Salinesi. 2014. "Towards a

Requirements Specification Multi- View Framework for Self-Adaptive Systems." *Computing Conference (CLEI), 2014 XL Latin American* 18 (2): 1–12. https://doi.org/10.13140/2.1.3132.8640.

Muñoz-fernández, Juan C, Gabriel Tamura, Irina Raicu, Raúl Mazo, and Camille Salinesi. 2015. "REFAS : A PLE Approach for Simulation of Self- Adaptive Systems Requirements." *Proceedings 19th International Software Product Line Conference (SPLC2015)*, no. September.

Neto, João José, Jorge Rady, and Almeida Júnior. 1999. "Modeling Adaptive Reactive Systems." In *International Conference Applied Modelling and Simulation*. Australia.

OMG. 2017. "Systems Modeling Language™ (OMG SysML™)." 2017. www.omgsysml.org.

Ortiz, Oscar, Ana Belen Garcia, Rafael Capilla, Jan Bosch, and Mike Hinchey. 2012. "Runtime Variability for Dynamic Reconfiguration in Wireless Sensor Network Product Lines." In *16th International Software Product Line Conference*, edited by ACM, Volume 2.

Parra, Carlos, Xavier Blanc, Anthony Cleve, and Laurence Duchien. 2011. "Unifying Design and Runtime Software Adaptation Using Aspect Models." *Science of Computer Programming* 76 (12): 1247–60. https://doi.org/10.1016/J.SCICO.2010.12.005.

Peña, Joaquin, Christopher A. Rouff, Mike Hinchey, and Antonio Ruiz-Cortés. 2011. "Modeling NASA Swarm-Based Systems: Using Agent-Oriented Software Engineering and Formal Methods." *Software & Systems Modeling* 10 (1): 55–62. https://doi.org/10.1007/s10270-009-0135-2.

Perera, Charith, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. 2014. "Context Aware Computing for The Internet of Things: A Survey." *IEEE Communications Surveys & Tutorials* 16 (1): 414–54. https://doi.org/10.1109/SURV.2013.042313.00197.

Pfannemueller, Martin, Christian Krupitzer, Markus Weckesser, and Christian Becker. 2017. "A Dynamic Software Product Line Approach for Adaptation Planning in Autonomic Computing Systems." In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, 247–54. IEEE. https://doi.org/10.1109/ICAC.2017.18.

Pimentel, João, Emanuel Santos, Jaelson Castro, and Xavier Franch. 2012. "Anticipating Requirements Changes-Using Futurology in Requirements Elicitation." *International Journal of Information System Modeling and Design* 3 (2): 89–111. https://doi.org/10.4018/jismd.2012040104.

Pohl, Klaus. 2010. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Incorporated.

Pohl, Klaus, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering. Foundations, Principles, and Techniques*. *Uwplatt.Edu*. Vol. 49. https://doi.org/10.1007/3-540-28901-1.

Quinton, Clément, Andreas Pleuss, Daniel Le Berre, Laurence Duchien, and Goetz Botterweck. 2014. "Consistency Checking for the Evolution of Cardinality-Based Feature Models." In *ACM International Conference Proceeding Series*, 1:122–31. New York, New York, USA: ACM Press. https://doi.org/10.1145/2648511.2648524.

Quinton, Clément, Daniel Romero, and Laurence Duchien. 2013. "Cardinality-Based Feature Models with Constraints: A Pragmatic Approach." In *ACM International Conference Proceeding Series*, 162–66. New York, New York, USA: ACM Press. https://doi.org/10.1145/2491627.2491638.

Qureshi, Nauman A., Ivan J. Jureta, and Anna Perini. 2011. "Requirements Engineering for Self-Adaptive

Systems: Core Ontology and Problem Statement." *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6741 LNCS: 33–47. https://doi.org/10.1007/978-3-642-21640-4_5.

Rajalakshmi, P., and S. Devi Mahalakshmi. 2016. "IOT Based Crop-Field Monitoring and Irrigation Automation." *Proceedings of the 10th International Conference on Intelligent Systems and Control, ISCO 2016*. https://doi.org/10.1109/ISCO.2016.7726900.

Reinhartz-Berger, Iris, Arnon Sturm, and Yair Wand. 2011. "External Variability of Software: Classification and Ontological Foundations." In *International Conference on Conceptual Modeling*, 6998), 275–89. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-24606-7_21.

Rolland, Colette. 2003. "Ingénierie Des Besoins : L'Approche L'Ecritoire." *Journal Techniques de l'Ingénieur*, 1.

Rossi, Francesca, Peter. Van Beek, and Toby. Walsh. 2006. *Handbook of Constraint Programming*. Elsevier.

Rouvoy, Romain, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli, and Ulrich Scholz. 2009. "MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments." *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5525 LNCS: 164–82. https://doi.org/10.1007/978-3-642-02161-9_9.

Sabatucci, Luca, Massimo Cossentino, and Valeria Seidita. 2018. "The Four Types of Self-Adaptive Systems: A Metamodel." *Intelligent Interactive Multimedia Systems and Services 2017*, no. May. https://doi.org/10.1007/978-3-319-59480-4.

Saleh, Majd, and Marie Hélène Abel. 2018. "System of Information Systems to Support Learners (a Case Study at the University of Technology of Compiègne)." *Behaviour and Information Technology* 37 (10–11): 1097–1110. https://doi.org/10.1080/0144929X.2018.1502808.

Salinesi, Camille, Raul Mazo, Daniel Diaz, and Olfa Djebbi. 2010. "Using Integer Constraint Solving in Reuse Based Requirements Engineering." In *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference, RE2010*, 243–51. IEEE. https://doi.org/10.1109/RE.2010.36.

Salinesi, Camille, Raul Mazo, Olfa Djebbi, Daniel Diaz, and Alberto Lora-Michiels. 2011. "Constraints: The Core of Product Line Engineering." In *Fifth International Conference On Research Challenges In Information Science*, 1–10. IEEE. https://doi.org/10.1109/RCIS.2011.6006825.

Salinesi, Camille, Clotilde Rohleder, Asmaa Achtaich, and Indra Kusumah. 2020. "Innocrowd, A Contribution to an IoT Based Engineering Product Development." In *International Conference on Computer Science and Information Technology*, 10:1–9. Zurich: AIRCC Publishing Corporation. https://doi.org/10.5121/CSIT.2020.100101.

Santos, Erick Barros dos, Rossana Maria de Castro Andrade, and Ismayle de Sousa Santos. 2019. "Runtime Monitoring of Behavioral Properties in Dynamically Adaptive Systems," 377–86. https://doi.org/10.1145/3350768.3351798.

Sawyer, Pete, Raul Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. 2012a. "Constraint Programming as a Means to Manage Configurations in Self-Adaptive Systems." *Special Issue in IEEE Computer Journal "Dynamic Software Product Lines"* 45 (October 2015): 56–63.

Sawyer, Pete, Raúl Mazo, Daniel Diaz, Camille Salinesi, and Danny Hughes. 2012b. "Using Constraint

Programming to Manage Configurations in Self-Adaptive Systems." *Computer* 45 (10): 56–63. https://doi.org/10.1109/MC.2012.286.

Schilit, Bill, Norman Adams, and Roy Want. 1994. "Context-Aware Computing Applications." In *Mobile Computing Systems and Applications*, 85–90.

Schneeweiss, Denny, and Petra Hofstedt. 2013. "FdConfig: A Constraint-Based Interactive Product Configurator." In *Lecture Notes in Computer Science*, 239–55. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-41524-1_13.

Semmak, Farida, Christophe Gnaho, and Régine Laleau. 2010. "Extended KAOS Method to Model Variability in Requirements." In *Communications in Computer and Information Science*, 193–205. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-14819-4_14.

Sewell, Alison, Tracey-Lynne Cody, Kama Weir, and Sally Hansen. 2018. "Innovations at the Boundary: An Exploratory Case Study of a New Zealand School-University Partnership in Initial Teacher Education." *Asia-Pacific Journal of Teacher Education* 46 (4): 321–39. https://doi.org/10.1080/1359866X.2017.1402294.

Sharifloo, Amir Molzam, Andreas Metzger, Clément Quinton, Luciano Baresi, and Klaus Pohl. 2016. "Learning and Evolution in Dynamic Software Product Lines." In *Proceedings - 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2016*, 158–64. New York, New York, USA: ACM Press. https://doi.org/10.1145/2897053.2897058.

Shenhar, Aaron J. 2001. "One Size Does Not Fit All Projects: Exploring Classical Contingency Domains." *Management Science*. INFORMS. https://doi.org/10.2307/2661507.

Sillitto, Hillary, James Martin, Dorothy Mckinney, Regina Griego, Dov Dori, Daniel Krob, Patrick Godfrey, Eileen Arnold, and Scott Jackson. 2019. "Systems Engineering and System Definitions."

Silva, Danyllo, Taisa Guidini Gonçalves, and Ana Regina C. da Rocha. 2019. "A Requirements Engineering Process for IoT Systems." In *Proceedings of the XVIII Brazilian Symposium on Software Quality - SBQS'19*, 204–9. New York, New York, USA: ACM Press. https://doi.org/10.1145/3364641.3364664.

Singh, Dhananjay, Gaurav Tripathi, and Antonio J. Jara. 2014. "A Survey of Internet-of-Things: Future Vision, Architecture, Challenges and Services." *2014 IEEE World Forum on Internet of Things, WF-IoT 2014*, no. MARCH 2014: 287–92. https://doi.org/10.1109/WF-IoT.2014.6803174.

Soares, Monique, Vilela Jéssyka, Gabriela Guedes, Carla Silva, and Jaelson Castro. 2017. "Core Ontology to Aid the Goal Oriented Specification for Self-Adaptive Systems." *Advances in Intelligent Systems and Computing* 571: V–VI. https://doi.org/10.1007/978-3-319-31232-3.

Solingen, Rini van, Vic Basili, Gianluigi Caldiera, and H. Dieter Rombach. 2002. "Goal Question Metric (GQM) Approach." In *Encyclopedia of Software Engineering*. Hoboken, NJ, USA: John Wiley & Sons, Inc. https://doi.org/10.1002/0471028959.sof142.

Sommerville, Ian. 2015. *Software Engineering*. Pearson Education Limited.

Sousa, Amanda, Anderson Uchôa, Eduardo Fernandes, Carla I M Bezerra, José Maria Monteiro, and Rossana M C Andrade. 2019. "REM4DSPL: A Requirements Engineering Method for Dynamic Software Product Lines." In *Proceedings of the XVIII Brazilian Symposium on Software Quality*, 10:129–38. ACM. https://doi.org/10.1145/3364641.3364656.

Souza, Vítor E. S. 2012. "Requirements-Based Software System Adaptation," no. June. http://eprints-

phd.biblio.unitn.it/809/.

Souza, Vítor E. S, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. 2013. "Awareness Requirements." *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7475 LNCS: 133–61. https://doi.org/10.1007/978-3-642-35813-5_6.

Souza, Vítor E.Silva, Alexei Lapouchnian, and John Mylopoulos. 2012. "(Requirement) Evolution Requirements for Adaptive Systems." In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 155–64. https://doi.org/10.1109/SEAMS.2012.6224402.

Streitferdt, Detlef, Sochos Periklis, Heller Christian, and Philippow Ilka. 2005. "Configuring Embedded System Families Using Feature Models." In *In Proc. of Net. ObjectDays* , 339–50.

Tesei, Luca, Emanuela Merelli, and Nicola Paoletti. 2013. "Multiple Levels in Self-Adaptive Complex Systems: A State-Based Approach." In *European Conference on Complex Systems.* Cham: Springer.

Tomer, Amir, and Stephen R Schach. 2000. "The Evolution Tree: A Maintenance-Oriented Software Development Model." In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, 209–14. IEEE Comput. Soc. https://doi.org/10.1109/CSMR.2000.827329.

Triki, Raouia, Camille Salinesi, and Raul Mazo. 2015. "Three Strategies to Specify Multi-Instantiation in Product Lines." In *Proceedings - International Conference on Research Challenges in Information Science*, 2015–June:211–16. IEEE Computer Society. https://doi.org/10.1109/RCIS.2015.7128882.

Tueno, Steve, Régine Laleau, Amel Mammar, and Marc Frappier. 2017. "The SysML/KAOS Domain Modeling Approach," October.

Turner, Mark, Rumjit Kaur, and Pearl Brereton. 2008. "A Lightweight Systematic Literature Review of Studies about the Use of Pair Programming to Teach Introductory Programming." In *Psychology of Programming Interest*. https://www.researchgate.net/publication/228681971_A_Lightweight_Systematic_Literature_Review_of_Studies_about_the_use_of_Pair_Programming_to_Teach_Introductory_Programming.

Union, International Telecommunication. 2012. "Overview of the Internet of Things." *Recommendation ITU-T*.

Uthariaraj, V Rhymend, and P Mercy Florence. 2011. "QoS With Reliability And Scalability In Adaptive Service-Based Systems," 37–56.

Vassev, Emil. 2015. "Requirements Engineering for Self-Adaptive Systems with ARE and KnowLang." *EAI Endorsed Transactions on Self-Adaptive Systems* 1 (1): e6. https://doi.org/10.4108/sas.1.1.e6.

Vassev, Emil, and Mike Hinchey. 2015. "Engineering Requirements for Autonomy Features." *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8998: 379–403. https://doi.org/10.1007/978-3-319-16310-9_11.

Villota, Angela, Raúl Mazo, and Camille Salinesi. 2018. "On the Ontological Expressiveness of the High-Level Constraint Language for Product Line Specification." In *System Analysis and Modeling. Languages, Methods, and Tools for Systems Engineering*, 46–66. Springer, Cham. https://doi.org/10.1007/978-3-030-01042-3_4.

Voelter, Markus, and Iris Groher. 2007. "Product Line Implementation Using Aspect-Oriented and Model-Driven Software Development." In *11th International Software Product Line Conference (SPLC 2007)*,

233–42. IEEE. https://doi.org/10.1109/SPLINE.2007.23.

Waaijenberg, Dries. 2006. “Design, Construction and Maintenance of Greenhouse Structures.” In *Acta Horticulturae*, 710:31–42. International Society for Horticultural Science. https://doi.org/10.17660/ActaHortic.2006.710.1.

Weyns, Danny, and Jesper Andersson. 2013. “On the Challenges of Self-Adaptation in Systems of Systems.” In *1st ACM SIGSOFT/SIGPLAN International Workshop on Software Engineering for Systems-of-Systems, SESoS 2013 Proceedings*, 47–51. https://doi.org/10.1145/2489850.2489860.

Weyns, Danny, and Bartosz Michalik. 2011. “Codifying Architecture Knowledge to Supportonline Evolution of Software Product Lines.” In *Proceeding of the 6th International Workshop on SHAring and Reusing Architectural Knowledge - SHARK ’11*, 37. New York, New York, USA: ACM Press. https://doi.org/10.1145/1988676.1988684.

Whittle, Jon, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. 2010. “RELAX: A Language to Address Uncertainty in Self-Adaptive Systems Requirement.” *Requirements Engineering* 15 (2): 177–96. https://doi.org/10.1007/s00766-010-0101-0.

Wolter, Katharina, Lothar Hotz, and Thorsten Krebs. 2006. “Model-Based Configuration Support For Software Product Families.” In *Mass Customization: Challenges and Solutions*, 43–61. Boston: Kluwer Academic Publishers. https://doi.org/10.1007/0-387-32224-8_3.

Xu, Li Da, Wu He, and Shancang Li. 2014. “Internet of Things in Industries: A Survey.” *IEEE Transactions on Industrial Informatics* 10 (4): 2233–43. https://doi.org/10.1109/TII.2014.2300753.

Yang, Qi Liang, Jian Lv, Xian Ping Tao, Xiao Xing Ma, Jian Chun Xing, and Wei Song. 2013. “Fuzzy Self-Adaptation of Mission-Critical Software under Uncertainty.” *Journal of Computer Science and Technology* 28 (1): 165–87. https://doi.org/10.1007/s11390-013-1321-9.

Yu, E.S.K. 1997. “Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering.” In *Proceedings of ISRE ’97: 3rd IEEE International Symposium on Requirements Engineering*, 226–35. IEEE Comput. Soc. Press. https://doi.org/10.1109/ISRE.1997.566873.

Zhang, Xueyan, Jianwu Zhang, Lin Li, Yuzhu Zhang, and Guocai Yang. 2017. “Monitoring Citrus Soil Moisture and Nutrients Using an IoT Based System.” *Sensors* 17 (3): 447. https://doi.org/10.3390/s17030447.

Annexe A - Recherche systématique : Echantillon Aléatoire 1

ID	Titre de l'article	Référence
1	(Requirement) Evolution Requirements for Adaptive Systems.	(Souza et al. 2012)
2	Runtime Monitoring of Behavioral Properties in Dynamically Adaptive Systems	(dos Santos et al. 2019)
3	Requirements Engineering for Self-Adaptive Systems: Core Ontology and Problem Statement	(Qureshi et al. 2011)
4	Fuzzy Self-Adaptation of Mission-Critical Software under Uncertainty	(Yang et al. 2013)
5	Requirements-Driven Social Adaptation: Expert Survey	(Almaliki et al. 2014)
6	Software Engineering of Self-Adaptive Systems: An Organised Tour and Future Challenges.	(Weyns 2017)
7	Law and Adaptivity in Requirements Engineering	(Ingolfo and Souza 2013)
8	Ontological Foundations for Software Requirements with a Focus on Requirements at Runtime	(Duarte et al. 2018)
9	Engineering Requirements for Autonomy Features	(Vassev and Hinchey 2015)
10	Integrating Goal Models and Problem Frames for Requirements Analysis of Self-Adaptive CPS	(Han et al. 2017)
11	On the Role of Model-Driven Engineering in Adaptive Systems	(Bocanegra et al. 2016)
12	A Framework to Support Consistent Design and Evolution of Adaptive Systems	(Mori 2012)
13	Anticipating Requirements Changes-Using Futurology in Requirements Elicitation	(Pimentel et al. 2012)
14	10 Challenges for the Specification of Self-Adaptive Software	(Muñoz-Fernández et al. 2018)
15	Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time	(Becker et al. 2013)
16	Towards an Ontology of Requirements at Runtime.	(Duarte et al. 2016)
17	An Agent-Based Self-Adaptation Architecture for Implementing Smart Devices in Smart Space.	(Chun et al. 2013)
18	Comparison of Approaches for Developing Self-Adaptive Systems.	(Krupitzer et al. 2018)
19	Capturing Ambiguity in Artifacts to Support Requirements Engineering for Self-Adaptive Systems.	(Muñoz-Fernández et al. 2017)
20	Awareness Requirements	(Souza et al. 2013)

21	Straightforward Specification of Adaptation-Architecture-Significant Requirements of IoT-Enabled Cyber-Physical Systems	(Antonino et al. 2018)a
22	Autonomic Software Product Lines (ASPL)	(Abbas et al. 2010)
23	Towards Requirements Engineering Process for Self-adaptive Embedded Systems	(Mecibah and Boutekkouk 2019)
24	The Four Types of Self-Adaptive Systems: a Metamodel	(Sabatucci et al. 2018)
25	Requirements Engineering for Self-Adaptive Systems with ARE and KnowLang	(Vassev 2015)
26	RELAX: A Language to Address Uncertainty in Self-Adaptive Systems Requirement	(Whittle et al. 2010)
27	Core Ontology to Aid the Goal Oriented Specification for Self-Adaptive Systems	(Soares et al. 2017)
28	QoS With Reliability And Scalability In Adaptive Service-Based Systems	(Uthariaraj and Florence 2011)
29	Requirements-Based Software System Adaptation	(Souza 2012)
30	Towards a Requirements Specification Multi- View Framework for Self-Adaptive Systems	(Muñoz-Fernández et al. 2014)

Annexe B : Modèle SCT – Flotte Maria

```
stateMachine ThesisExample
//The greenLife Fleet elements
Boolean Fleet
Boolean Actuator
Boolean Sprinkler [1,9]
Boolean Memory [1,9]
Integer Enum Rotation [1,9]{0, 45, 90, 180, 240, 360}
Boolean Head [1,9]
Integer PressureSp [1,9] [0..60]
Boolean Rotor [1,9]
Boolean Spray [1,9]
Boolean CenterPivor
Integer Enum EmitterCP {0, 114,232}
Integer PressureCP [0..60]
Boolean SprinklerCP
Boolean Dripline
Integer Enum Emitter {0, 114,232}
Integer PressureDr [0..60]
Boolean Humidifier
Boolean Rooftop
Boolean Position
Boolean Param IN_open
Boolean Param IN_close
Boolean OUT_open
Boolean OUT_close
Boolean Partial
Boolean Roof
Boolean Shader
Boolean Glass
Boolean LightBulb [1,5]
Boolean MistDispenser
Boolean MistTemp
Boolean PressureMD
Boolean Nozzle
Boolean AirConditionner [1,4]
Boolean AirMode[1,4][0..3] // 0=nonSelected 1=Fan 2=cool 3=Heating
Integer Temp[1,4][0..30]
Boolean Speed[1,4][0..3]  //0=nonSelected 1=low 2=medium 3=high
Boolean Sensor
Boolean RainSensor
Integer Enum TimeSpan {0, 10}
Boolean TempSensor
Boolean LightSensor
```

```
Boolean HumiditySensor[1,7]
Boolean Notifier[1,7]
Boolean Alarm[1,7]
Boolean Contact[1,7]
Boolean Bulb[1,7]
Boolean Mode[1,7]
Boolean Master[1,7]
Boolean Slave[1,7]
Boolean HumidityType[1,7]
Boolean Air[1,7]
Boolean Soil[1,7]
Boolean SmartMeter
Boolean WaterSupplier
Boolean TreasholdAlam
Boolean Source
Boolean Fertilizer
Boolean Pump
Boolean RainWater
Boolean Mainswater
Boolean Communication
Boolean Wifi
Boolean Zigbee
Boolean Router
Boolean Coordinator
Boolean Topology
Boolean Distributed
Boolean Star

//Context variables
Float Param BatteryLevel[1,7]
Integer Param Temperature
Float Param Brightness
Float Param minBrightness
Float Param opBrightness
Float Param TankLevel
Float Param MinTankWater
Float Param OpTankWater
Float Param WaterConsumption
Boolean Param Rain
Float Param EConsumption
Float Param OpConsumption
Boolean Param PowerFailure
Integer Param Month
Float Param HumidityLevel
Float Param NormalHumidity

//*******************FLEXIBLE CONSTRAINTS*****************
```

```
//----------The main capabilities
Fleet=Communication
Fleet=Actuator
Fleet=Sensor
Fleet=WaterSupplier
//----------Communication
Communication=Zigbee
Communication>=Wifi
Zigbee=Coordinator
Zigbee>=Router
Zigbee>=Topology
Topology=Distributed+Star
//----------Actuator
Actuator>=Rooftop
Rooftop=Position
Rooftop=Roof
Position=OUT_open+OUT_close
Position>=Partial
Roof=Glass
Roof>=Shader
Actuator>=MistDispenser
MistDispenser >0 -> MistTemp>0
MistDispenser = Nozzle
MistDispenser>0 -> PressureMD >0
Actuator>=Humidifier
Actuator>=LightBulb[1] /\ Actuator>=LightBulb[2] /\ Actuator>=LightBulb[3] /\ Actuator>=LightBulb[4]
/\ Actuator>=LightBulb[5] /\ Actuator=1 -> ((LightBulb[1]+ LightBulb[2]+ LightBulb[3]+ LightBulb[4]+
LightBulb[5])>0 /\ (LightBulb[1]+ LightBulb[2]+ LightBulb[3]+ LightBulb[4]+ LightBulb[5])<5)
AirConditionner[1]>0 -> AirMode[1]>0 /\ AirConditionner[1]>0 -> Speed[1]>0 /\ AirConditionner[1]>0 ->
Temp[1]>0
AirConditionner[2]>0 -> AirMode[2]>0 /\ AirConditionner[2]>0 -> Speed[2]>0 /\ AirConditionner[2]>0 ->
Temp[2]>0
AirConditionner[3]>0 -> AirMode[3]>0 /\ AirConditionner[3]>0 -> Speed[3]>0 /\ AirConditionner[3]>0 ->
Temp[3]>0
AirConditionner[4]>0 -> AirMode[4]>0 /\ AirConditionner[4]>0 -> Speed[4]>0 /\ AirConditionner[4]>0 ->
Temp[4]>0
Actuator>=0                                                                                    ->
(Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler[6]+Sprinkler[7]+Sprinkle
r[8]+Sprinkler[9])*CenterPivor=                                    0                           /\
(Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler[6]+Sprinkler[7]+Sprinkle
r[8]+Sprinkler[9])*Dripline=0              /\              (Dripline*CenterPivor=0)             /\
(Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler[6]+Sprinkler[7]+Sprinkle
r[8]+Sprinkler[9] + Dripline + CenterPivor >0)
Sprinkler[1]>0 -> Memory[1]>0 /\ Sprinkler[1]>0 -> Rotation[1]>0 /\ Sprinkler[1]>0 -> PressureSp [1]>0
/\ Sprinkler[1]>0 -> Head [1]>0
```

```
Sprinkler[2]>0 -> Memory[2]>0 /\ Sprinkler[2]>0 -> Rotation[2]>0 /\ Sprinkler[2]>0 -> PressureSp [2]>0
/\ Sprinkler[1]>0 -> Head [2]>0
Sprinkler[3]>0 -> Memory[3]>0 /\ Sprinkler[3]>0 -> Rotation[3]>0 /\ Sprinkler[3]>0 -> PressureSp [3]>0
/\ Sprinkler[1]>0 -> Head [3]>0
Sprinkler[4]>0 -> Memory[4]>0 /\ Sprinkler[4]>0 -> Rotation[4]>0 /\ Sprinkler[4]>0 -> PressureSp [4]>0
/\ Sprinkler[1]>0 -> Head [4]>0
Sprinkler[5]>0 -> Memory[5]>0 /\ Sprinkler[5]>0 -> Rotation[5]>0 /\ Sprinkler[5]>0 -> PressureSp [5]>0
/\ Sprinkler[5]>0 -> Head [5]>0
Sprinkler[6]>0 -> Memory[6]>0 /\ Sprinkler[6]>0 -> Rotation[6]>0 /\ Sprinkler[6]>0 -> PressureSp [6]>0
/\ Sprinkler[6]>0 -> Head [6]>0
Sprinkler[7]>0 -> Memory[7]>0 /\ Sprinkler[7]>0 -> Rotation[7]>0 /\ Sprinkler[7]>0 -> PressureSp [7]>0
/\ Sprinkler[7]>0 -> Head [7]>0
Sprinkler[8]>0 -> Memory[8]>0 /\ Sprinkler[8]>0 -> Rotation[8]>0 /\ Sprinkler[8]>0 -> PressureSp [8]>0
/\ Sprinkler[8]>0 -> Head [8]>0
Sprinkler[9]>0 -> Memory[9]>0 /\ Sprinkler[9]>0 -> Rotation[9]>0 /\ Sprinkler[9]>0 -> PressureSp [9]>0
/\ Sprinkler[9]>0 -> Head [9]>0
Head[1] = Spray[1]+ Rotor[1]
Head[2] = Spray[2]+ Rotor[2]
Head[3] = Spray[3]+ Rotor[3]
Head[4] = Spray[4]+ Rotor[4]
Head[5] = Spray[5]+ Rotor[5]
Head[6] = Spray[6]+ Rotor[6]
Head[7] = Spray[7]+ Rotor[7]
Head[8] = Spray[8]+ Rotor[8]
Head[9] = Spray[9]+ Rotor[9]
CenterPivor=SprinklerCP
CenterPivor >0 -> EmitterCP>0
CenterPivor >0 -> PressureCP>0
Dripline >0 -> Emitter>0
Dripline >0 -> PressureDr>0
//----------Sensors
Sensor>=SmartMeter
Sensor>=RainSensor
RainSensor>0 -> TimeSpan>0
Sensor>=LightSensor
Sensor>=TempSensor
Sensor>=SmartMeter
Sensor>=HumiditySensor[1]    /\    Sensor>=HumiditySensor[2]    /\    Sensor>=HumiditySensor[3]    /\
Sensor>=HumiditySensor[4]    /\    Sensor>=HumiditySensor[5]    /\       Sensor>=HumiditySensor[6]    /\
Sensor>=HumiditySensor[7] /\ Sensor=1 -> ((HumiditySensor[1]+ HumiditySensor[2]+ HumiditySensor[3]+
HumiditySensor[4]+ HumiditySensor[5] + HumiditySensor[6]+ HumiditySensor[7]>=1) /\ (HumiditySensor[1]+
HumiditySensor[2]+  HumiditySensor[3]+  HumiditySensor[4]+  HumiditySensor[5]  +  HumiditySensor[6]+
HumiditySensor[7]<=7))
HumiditySensor[1] >= Notifier[1] /\ HumiditySensor[1] = Mode[1] /\ HumiditySensor[1]= HumidityType[1]
/\ HumidityType[1] = Soil[1] /\ HumidityType[1] >= Air[1] /\ Notifier[1] * 3 >= Alarm[1] + Contact[1]
+ Bulb[1] /\ Mode[1]= Slave[1] + Master[1]
```

```
HumiditySensor[2] >= Notifier[2] /\ HumiditySensor[2] = Mode[2] /\ HumiditySensor[2]= HumidityType[2]
/\ HumidityType[2] = Soil[2] /\ HumidityType[2] >= Air[2] /\ Notifier[2] * 3 >= Alarm[2] + Contact[2]
+ Bulb[2] /\ Mode[2]= Slave[2] + Master[2]
HumiditySensor[3] >= Notifier[3] /\ HumiditySensor[3] = Mode[3] /\ HumiditySensor[3]= HumidityType[3]
/\ HumidityType[3] = Soil[3] /\ HumidityType[3] >= Air[3] /\ Notifier[3] * 3 >= Alarm[3] + Contact[3]
+ Bulb[3] /\ Mode[3]= Slave[3] + Master[3]
HumiditySensor[4] >= Notifier[4] /\ HumiditySensor[4] = Mode[4] /\ HumiditySensor[4]= HumidityType[4]
/\ HumidityType[4] = Soil[4] /\ HumidityType[4] >= Air[4] /\ Notifier[4] * 3 >= Alarm[4] + Contact[4]
+ Bulb[4] /\ Mode[4]= Slave[4] + Master[4]
HumiditySensor[5] >= Notifier[5] /\ HumiditySensor[5] = Mode[5] /\ HumiditySensor[5]= HumidityType[5]
/\ HumidityType[5] = Soil[5] /\ HumidityType[5] >= Air[5] /\ Notifier[5] * 3 >= Alarm[5] + Contact[5]
+ Bulb[5] /\ Mode[5]= Slave[5] + Master[5]
HumiditySensor[6] >= Notifier[6] /\ HumiditySensor[6] = Mode[6] /\ HumiditySensor[6]= HumidityType[6]
/\ HumidityType[6] = Soil[6] /\ HumidityType[6] >= Air[6] /\ Notifier[6] * 3 >= Alarm[6] + Contact[6]
+ Bulb[6] /\ Mode[6]= Slave[6] + Master[6]
HumiditySensor[7] >= Notifier[7] /\ HumiditySensor[7] = Mode[7] /\ HumiditySensor[7]= HumidityType[7]
/\ HumidityType[7] = Soil[7] /\ HumidityType[7] >= Air[7] /\ Notifier[7] * 3 >= Alarm[7] + Contact[7]
+ Bulb[7] /\ Mode[7]= Slave[7] + Master[7]
//----------Water Supply
WaterSupplier>=TreasholdAlam
WaterSupplier>=Fertilizer
WaterSupplier>=Pump
WaterSupplier=Source
Source=RainWater+Mainswater
//---------Traversal
RainWater>0 -> Pump>0
RainWater>0 -> TreasholdAlam>0
Mainswater>0 -> SmartMeter>0
Humidifier>0 -> (Air[1]+Air[2]+Air[3]+Air[4]+Air[5]+Air[6]+Air[7]>0)
OUT_open * (LightBulb[1]+ LightBulb[2]+ LightBulb[3]+ LightBulb[4]+ LightBulb[5]) =0

//********************************** States ***********************************

compositeState AW
        concernLevel global_AW
                simpleConstraintState global_AW_state
                        CenterPivor=0
                        Humidifier=0
                        MistDispenser=0
                        Fertilizer=0
                        (Air[1]+Air[2]+Air[3]+Air[4]+Air[5]+Air[6]+Air[7]=0)
                        Router=0
                        Distributed=0
                        Shader=0
                        Actuator>0                                                      ->
                        (((Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler
                        [6]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9])*Dripline          =0)          /\
```

```
(Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+
Sprinkler[5]+Sprinkler[6]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9]  +  Dripline
>=               0)               /\((Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+
Sprinkler[4]+Sprinkler[5]+Sprinkler[6]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9]
>=0                                  /\Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+
Sprinkler[4]+Sprinkler[5]+Sprinkler[6]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9]
<=4) \/ Dripline=1))
Actuator>0                                                                  ->
(AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4]>
=0                                   /\AirConditionner[1]+AirConditionner[2]+
AirConditionner[3]+AirConditionner[4] <= 4)
Actuator>0   ->   (LightBulb[1]+   LightBulb[2]+   LightBulb[3]+   LightBulb[4]+
LightBulb[5] >=0 /\ LightBulb[1]+ LightBulb[2]+ LightBulb[3]+ LightBulb[4]+
LightBulb[5] <=5)
HumiditySensor[1]   >=   Notifier[1]   /\   HumiditySensor[1]   =   Mode[1]   /\
HumiditySensor[1]=   HumidityType[1]   /\   HumidityType[1]   =   Soil[1]   /\
HumidityType[1] >= Air[1] /\ ((Notifier[1]>0) ->   (Alarm[1] + Contact[1] +
Bulb[1]<=3)) /\ Mode[1]= Slave[1] + Master[1]
HumiditySensor[2]   >=   Notifier[2]   /\   HumiditySensor[2]   =   Mode[2]   /\
HumiditySensor[2]=   HumidityType[2]   /\   HumidityType[2]   =   Soil[2]   /\
HumidityType[2] >= Air[2] /\ ((Notifier[2]>0) ->   (Alarm[2] + Contact[2] +
Bulb[2]<=3)) /\ Mode[2]= Slave[2] + Master[2]
HumiditySensor[3]   >=   Notifier[3]   /\   HumiditySensor[3]   =   Mode[3]   /\
HumiditySensor[3]=   HumidityType[3]   /\   HumidityType[3]   =   Soil[3]   /\
HumidityType[3] >= Air[3] /\ ((Notifier[3]>0) ->   (Alarm[3] + Contact[3] +
Bulb[3]<=3)) /\ Mode[3]= Slave[3] + Master[3]
HumiditySensor[4]   >=   Notifier[4]   /\   HumiditySensor[4]   =   Mode[4]   /\
HumiditySensor[4]=   HumidityType[4]   /\   HumidityType[4]   =   Soil[4]   /\
HumidityType[4] >= Air[4] /\ ((Notifier[4]>0) ->   (Alarm[4] + Contact[4] +
Bulb[4]<=3)) /\ Mode[4]= Slave[4] + Master[4]
HumiditySensor[5]   >=   Notifier[5]   /\   HumiditySensor[5]   =   Mode[5]   /\
HumiditySensor[5]=   HumidityType[5]   /\   HumidityType[5]   =   Soil[5]   /\
HumidityType[5] >= Air[5] /\ ((Notifier[5]>0) ->   (Alarm[5] + Contact[5] +
Bulb[5]<=3)) /\ Mode[5]= Slave[5] + Master[5]
HumiditySensor[6]   >=   Notifier[6]   /\   HumiditySensor[6]   =   Mode[6]   /\
HumiditySensor[6]=   HumidityType[6]   /\   HumidityType[6]   =   Soil[6]   /\
HumidityType[6] >= Air[6] /\ ((Notifier[6]>0) ->   (Alarm[6] + Contact[6] +
Bulb[6]<=3)) /\ Mode[6]= Slave[6] + Master[6]
HumiditySensor[7]   >=   Notifier[7]   /\   HumiditySensor[7]   =   Mode[7]   /\
HumiditySensor[7]=   HumidityType[7]   /\   HumidityType[7]   =   Soil[7]   /\
HumidityType[7] >= Air[7] /\ ((Notifier[7]>0) ->   (Alarm[7] + Contact[7] +
Bulb[7]<=3)) /\ Mode[7]= Slave[7] + Master[7]
HumiditySensor[7]   >=   Notifier[7]   /\   HumiditySensor[7]   =   Mode[7]   /\
HumiditySensor[7]=   HumidityType[7]   /\   HumidityType[7]   =   Soil[7]   /\
HumidityType[7] >= Air[7] /\ Notifier[7] * 7 >= Alarm[7] + Contact[7] + Bulb[7]
/\ Mode[7]= Slave[7] + Master[7]
```

```
		Master[1]+Master[2]+Master[3]+Master[4]+Master[5]+Master[6]+Master[7]>=1
		\/Master[1]+Master[2]+Master[3]+Master[4]+Master[5]+ Master[6]+Master[7]<=3
		Sensor=LightSensor
		Sensor=TempSensor
		Sensor>=RainSensor
		Sensor>=SmartMeter
		Head[1]>0 -> Spray[1]>0
		Head[2]>0 -> Spray[2]>0
		Head[3]>0 -> Spray[3]>0
		Head[4]>0 -> Spray[4]>0
		Head[5]>0 -> Spray[5]>0
		Head[6]>0 -> Spray[6]>0
		Head[7]>0 -> Spray[7]>0
		Head[8]>0 -> Spray[8]>0
		Head[9]>0 -> Spray[9]>0
end_concernLevel global_AW

concernLevel irrigation_AW
	simpleConstraintState Natural_Irrigation_AW_state
		Dripline=0
		(Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+
		Sprinkler[5]+Sprinkler[6]+Sprinkler[7]+Sprinkler[8]+ Sprinkler[9])=0
		OUT_open=1
	compositeState automatic_AW
		concernLevel automatic_CL_AW
			simpleConstraintState rapidIrrigation_AW
				((Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+
				Sprinkler[4]+Sprinkler[5]+Sprinkler[6]+Sprinkler[7]+
				Sprinkler[8]+Sprinkler[9])=4)
				Sprinkler[1]+Sprinkler[3]+Sprinkler[7]+ Sprinkler[9] =4
				Rotation[1]=240
				Rotation[3]=240
				Rotation[7]=240
				Rotation[9]=240
				PressureSp[1]=30
				PressureSp[3]=30
				PressureSp[7]=30
				PressureSp[9]=30
			simpleConstraintState slowIrrigation_AW
				Dripline=1
				Emitter=114
		end_concernLevel automatic_CL_AW
	end_compositeState automatic_AW
end_concernLevel irrigation_AW

concernLevel heating_AW
	simpleConstraintState normal_heating_AW_state
```

```
                LightBulb[1]=1 /\ LightBulb[3]=1 /\ LightBulb[5]=1
                LightBulb[1]+LightBulb[2]+LightBulb[3]+LightBulb[4]+ LightBulb[5]=3
        simpleConstraintState high_heating_AW_state
                LightBulb[1]+LightBulb[2]+LightBulb[3]+LightBulb[4]+ LightBulb[5]=5
        simpleConstraintState open_air_AW_state
                AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+
                AirConditionner[4]=4
                Temp[1]=30 /\ Temp[2]=30 /\ Temp[3]=30 /\ Temp[4]=30
                Speed[1]=3 /\ Speed[2]=3 /\ Speed[3]=3 /\ Speed[4]=3
                AirMode[1]=3 /\ AirMode[2]=3 /\AirMode[3]=3 /\AirMode[4]=3
        simpleConstraintState extra_heating_AW_state
                AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+
                AirConditionner[4]=2
                AirConditionner[1]+AirConditionner[3]=2
                Temp[1]=30  /\ Temp[3]=30
                Speed[1]=2 /\ Speed[3]=2
                AirMode[1]=3 /\ AirMode[3]=3
end_concernLevel heating_AW

concernLevel moitSensing_AW
        simpleConstraintState acurate_moist_AW_state
                Master[1]+Master[2]+Master[3]+Master[4]+Master[5]+Master[6]+ Master[7]=3
        simpleConstraintState efficient_moit_AW_state
                Master[1]+Master[2]+Master[3]+Master[4]+Master[5]+Master[6]+  Master[7]=1
                BatteryLevel[1]<=40 -> Master[1]=0
                BatteryLevel[2]<=40 -> Master[2]=0
                BatteryLevel[3]<=40 -> Master[3]=0
                BatteryLevel[4]<=40 -> Master[4]=0
                BatteryLevel[5]<=40 -> Master[5]=0
                BatteryLevel[6]<=40 -> Master[6]=0
                BatteryLevel[7]<=40 -> Master[7]=0
end_concernLevel moitSensing_AW

concernLevel brightneAW_AW
        simpleConstraintState natural_brightness_AW_state
                OUT_open =1
        simpleConstraintState automatic_brightness_AW_state
                OUT_close =1
                LightBulb[1]+LightBulb[2]+LightBulb[3]+LightBulb[4]+ LightBulb[5]>0
end_concernLevel brightneAW_AW

concernLevel waterSource_AW
        simpleConstraintState efficientWater_AW_state
                RainWater =1
        simpleConstraintState durableWater_AW_state
                Mainswater =1
end_concernLevel waterSource_AW
```

```
end_compositeState AW


compositeState SS
        concernLevel global_SS
                simpleConstraintState global_SS_state
                        CenterPivor=0
                        Humidifier=0
                        MistDispenser=0
                        Fertilizer=0
                        (Air[1]+Air[2]+Air[3]+Air[4]+Air[5]+Air[6]+Air[7]=0)
                        Router=0
                        Distributed=0
                        Shader=0
                        Actuator>0 ->
                        ((Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler[
                        6]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9])*Dripline =0)
                        /\(Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler
                        [6]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9] + Dripline >= 0)
                        /\((Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkle
                        r[6]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9]>=0                                  /\
                        Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler[6]
                        +Sprinkler[7]+Sprinkler[8]+Sprinkler[9]<=9) \/ Dripline=1)
                        Actuator>0                                                                      ->
                        (AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4]>
                        =0                                                                              /\
                        AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4]
                        <= 4)
                        Actuator>0  ->  (LightBulb[1]+  LightBulb[2]+  LightBulb[3]+  LightBulb[4]+
                        LightBulb[5] >=0 /\ LightBulb[1]+ LightBulb[2]+ LightBulb[3]+ LightBulb[4]+
                        LightBulb[5] <=3)
                        HumiditySensor[1]  >=  Notifier[1]  /\  HumiditySensor[1]  =  Mode[1]  /\
                        HumiditySensor[1]=  HumidityType[1]  /\  HumidityType[1]  =  Soil[1]  /\
                        HumidityType[1] >= Air[1] /\ ((Notifier[1]>0) -> (Alarm[1] + Bulb[1]<=2 /\
                        Contact[1]>0) ) /\ Mode[1]= Slave[1] + Master[1]
                        HumiditySensor[2]  >=  Notifier[2]  /\  HumiditySensor[2]  =  Mode[2]  /\
                        HumiditySensor[2]=  HumidityType[2]  /\  HumidityType[2]  =  Soil[2]  /\
                        HumidityType[2] >= Air[2] /\ ((Notifier[2]>0) -> (Alarm[2] + Bulb[2]<=2 /\
                        Contact[2]>0) ) /\ Mode[2]= Slave[2] + Master[2]
                        HumiditySensor[3]  >=  Notifier[3]  /\  HumiditySensor[3]  =  Mode[3]  /\
                        HumiditySensor[3]=  HumidityType[3]  /\  HumidityType[3]  =  Soil[3]  /\
                        HumidityType[3] >= Air[3] /\ ((Notifier[3]>0) -> (Alarm[3] + Bulb[3]<=2 /\
                        Contact[3]>0) ) /\ Mode[3]= Slave[3] + Master[3]
                        HumiditySensor[4]  >=  Notifier[4]  /\  HumiditySensor[4]  =  Mode[4]  /\
                        HumiditySensor[4]=  HumidityType[4]  /\  HumidityType[4]  =  Soil[4]  /\
                        HumidityType[4] >= Air[4] /\ ((Notifier[4]>0) -> (Alarm[4] + Bulb[4]<=2 /\
                        Contact[4]>0) ) /\ Mode[4]= Slave[4] + Master[4]
```

```
                HumiditySensor[5]  >=  Notifier[5]  /\  HumiditySensor[5]  =  Mode[5]  /\
                HumiditySensor[5]=  HumidityType[5]  /\  HumidityType[5]  =  Soil[5]  /\
                HumidityType[5] >= Air[5] /\ ((Notifier[5]>0) -> (Alarm[5] + Bulb[5]<=2 /\
                Contact[5]>0) ) /\ Mode[5]= Slave[5] + Master[5]
                HumiditySensor[6]  >=  Notifier[6]  /\  HumiditySensor[6]  =  Mode[6]  /\
                HumiditySensor[6]=  HumidityType[6]  /\  HumidityType[6]  =  Soil[6]  /\
                HumidityType[6] >= Air[6] /\ ((Notifier[6]>0) -> (Alarm[6] + Bulb[6]<=2 /\
                Contact[6]>0) ) /\ Mode[6]= Slave[6] + Master[6]
                HumiditySensor[7]  >=  Notifier[7]  /\  HumiditySensor[7]  =  Mode[7]  /\
                HumiditySensor[7]=  HumidityType[7]  /\  HumidityType[7]  =  Soil[7]  /\
                HumidityType[7] >= Air[7] /\ ((Notifier[7]>0) -> (Alarm[7] + Bulb[7]<=2 /\
                Contact[7]>0) ) /\ Mode[7]= Slave[7] + Master[7]
                Master[1]+Master[2]+Master[3]+Master[4]+Master[5]+Master[6]+Master[7]>=1   \/
                Master[1]+Master[2]+Master[3]+Master[4]+Master[5]+Master[6]+Master[7]<=7
                Sensor>=LightSensor
                Sensor=TempSensor
                Sensor>=RainSensor
                Sensor=SmartMeter
                WaterSupplier=TreasholdAlam
end_concernLevel global_SS

concernLevel irrigation_SS
        simpleConstraintState Natural_Irrigation_SS_state
                Dripline=0
                (Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler[6
                ]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9])=0
        compositeState automatic_SS
                concernLevel automatic_CL_SS
                        simpleConstraintState rapidAcc_Irrigation_SS
                                (Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkle
                                r[5]+Sprinkler[6]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9])=9
                                (Rotor[1]+Rotor[2]+Rotor[3]+Rotor[4]+Rotor[5]+Rotor[6]+Rotor[
                                7]+Rotor[8]+Rotor[9])=9
                                (Rotation[1]=45  /\  Rotation[2]=45  /\  Rotation[3]=45  /\
                                Rotation[4]=45  /\  Rotation[5]=45  /\  Rotation[6]=45  /\
                                Rotation[7]=45 /\ Rotation[8]=45 /\ Rotation[9]=45)
                                (PressureSp[1]=60 /\ PressureSp[2]=60 /\ PressureSp[3]=60 /\
                                PressureSp[4]=60 /\ PressureSp[5]=60 /\ PressureSp[6]=60 /\
                                PressureSp[7]=60 /\ PressureSp[8]=60 /\ PressureSp[9]=60)
                        simpleConstraintState rapidEff_Irrigation_SS
                                (Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkle
                                r[5]+Sprinkler[6]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9])=4
                                /\ Sprinkler[1]+Sprinkler[3]+Sprinkler[7]+Sprinkler[9] =4
                                (Rotor[1]+Rotor[2]+Rotor[3]+Rotor[4]+Rotor[5]+Rotor[6]+Rotor[
                                7]+Rotor[8]+Rotor[9])=4
                                (Rotation[1]=240  /\   Rotation[3]=240  /\  Rotation[7]=240  /\
                                Rotation[9]=240 )
```

```
                        (PressureSp[1]=30 /\ PressureSp[3]=30 /\ PressureSp[7]=30 /\
                        PressureSp[9]=30)
                simpleConstraintState slowIrrigation_SS
                        Dripline=1
                        Emitter=232
        end_concernLevel automatic_CL_SS
    end_compositeState automatic_SS
end_concernLevel irrigation_SS

concernLevel heating_SS
    simpleConstraintState natural_cooling_SS_state
        AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4]=0
    simpleConstraintState automatic_cooling_SS_state
        AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4]=1
        Temp[1]+Temp[2]+Temp[3]+Temp[4]=20 /\ Temp[1]*Temp[2]=0 /\ Temp[3]*Temp[4]=0
        /\ Temp[1]*Temp[3]=0 /\ Temp[2]*Temp[4]=0
        Speed[1]+Speed[2]+Speed[3]+Speed[4]=3      /\      Speed[1]*Speed[2]=0      /\
        Speed[3]*Speed[4]=0 /\ Speed[1]*Speed[3]=0 /\ Speed[2]*Speed[4]=0
        Mode[1]+Mode[2]+Mode[3]+Mode[4]=2 /\ Mode[1]*Mode[2]=0 /\ Mode[3]*Mode[4]=0 /\
        Mode[1]*Mode[3]=0 /\ Mode[2]*Mode[4]=0
end_concernLevel heating_SS

concernLevel moitSensing_SS
    simpleConstraintState acurate_moist_SS_state
        Master[1]+Master[2]+Master[3]+Master[4]+Master[5]+Master[6]+Master[7]=7
    simpleConstraintState efficient_moit_SS_state
        Master[1]+Master[2]+Master[3]+Master[4]+Master[5]+Master[6]+Master[7]=3
        BatteryLevel[1]<=20 -> Master[1]=0
        BatteryLevel[2]<=20 -> Master[2]=0
        BatteryLevel[3]<=20 -> Master[3]=0
        BatteryLevel[4]<=20 -> Master[4]=0
        BatteryLevel[5]<=20 -> Master[5]=0
        BatteryLevel[6]<=20 -> Master[6]=0
        BatteryLevel[7]<=20 -> Master[7]=0
end_concernLevel moitSensing_SS

concernLevel brightness_SS
    simpleConstraintState natural_brightness_SS_state
        OUT_open=1
    simpleConstraintState automatic_brightness_SS_state
        OUT_close =1
        LightBulb[1]+LightBulb[2]+LightBulb[3]+LightBulb[4]+LightBulb[5]=3
end_concernLevel brightness_SS

concernLevel waterSource_SS
    simpleConstraintState efficientWater_SS_state
        RainWater=1
```

```
                simpleConstraintState durableWater_SS_state
                        Mainswater =1
        end_concernLevel waterSource_SS
end_compositeState SS

compositeState NoE
        concernLevel global_NoE
                simpleConstraintState global_NoE_state
                        CenterPivor=0
                        Humidifier=0
                        MistDispenser=0
                        Fertilizer=0
                        (Air[1]+Air[2]+Air[3]+Air[4]+Air[5]+Air[6]+Air[7]=0)
                        Router=0
                        Distributed=0
                        Shader=0
                        SmartMeter=0
                        RainSensor=0
                        LightSensor=0
                        TempSensor=0
                        RainWater=0
                        Fertilizer=0
                        TreasholdAlam=0
                        Actuator>0 -> Dripline=1
                        (Sprinkler[1]+Sprinkler[2]+Sprinkler[3]+Sprinkler[4]+Sprinkler[5]+Sprinkler[6
                        ]+Sprinkler[7]+Sprinkler[8]+Sprinkler[9])=0
                        Actuator>0                                                              ->
                        (AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4]=
                        0                                                                       \/
                        AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4] =
                        1)
                        LightBulb[1]+LightBulb[2]+LightBulb[3]+LightBulb[4]+LightBulb[5]=0

                        HumiditySensor[1]  >=  Notifier[1]  /\  HumiditySensor[1]  =  Mode[1]  /\
                        HumiditySensor[1]=  HumidityType[1]  /\  HumidityType[1]  =  Soil[1]  /\
                        HumidityType[1] >= Air[1] /\ (Notifier[1]>0 -> Contact[1]=1) /\ Mode[1]=
                        Slave[1] + Master[1]
                        HumiditySensor[2]  >=  Notifier[2]  /\  HumiditySensor[2]  =  Mode[2]  /\
                        HumiditySensor[2]=  HumidityType[2]  /\  HumidityType[2]  =  Soil[2]  /\
                        HumidityType[2] >= Air[2] /\ (Notifier[2]>0 -> Contact[2]=1) /\ Mode[2]=
                        Slave[2] + Master[2]
                        HumiditySensor[3]  >=  Notifier[3]  /\  HumiditySensor[3]  =  Mode[3]  /\
                        HumiditySensor[3]=  HumidityType[3]  /\  HumidityType[3]  =  Soil[3]  /\
                        HumidityType[3] >= Air[3] /\ (Notifier[3]>0 -> Contact[3]=1) /\ Mode[3]=
                        Slave[3] + Master[3]
                        HumiditySensor[4]  >=  Notifier[4]  /\  HumiditySensor[4]  =  Mode[4]  /\
                        HumiditySensor[4]=  HumidityType[4]  /\  HumidityType[4]  =  Soil[4]  /\
```

```
                HumidityType[4] >= Air[4] /\ (Notifier[4]>0 -> Contact[4]=1) /\ Mode[4]=
                Slave[4] + Master[4]
                HumiditySensor[5] >= Notifier[5] /\ HumiditySensor[5] = Mode[5] /\
                HumiditySensor[5]= HumidityType[5] /\ HumidityType[5] = Soil[5] /\
                HumidityType[5] >= Air[5] /\ (Notifier[5]>0 -> Contact[5]=1) /\ Mode[5]=
                Slave[5] + Master[5]
                HumiditySensor[6] >= Notifier[6] /\ HumiditySensor[6] = Mode[6] /\
                HumiditySensor[6]= HumidityType[6] /\ HumidityType[6] = Soil[6] /\
                HumidityType[6] >= Air[6] /\ (Notifier[6]>0 -> Contact[6]=1) /\ Mode[6]=
                Slave[6] + Master[6]
                HumiditySensor[7] >= Notifier[7] /\ HumiditySensor[7] = Mode[7] /\
                HumiditySensor[7]= HumidityType[7] /\ HumidityType[7] = Soil[7] /\
                HumidityType[7] >= Air[7] /\ (Notifier[7]>0 -> Contact[7]=1) /\ Mode[7]=
                Slave[7] + Master[7]
                (Alarm[1] + Contact[1] + Bulb[1] = 1)
                (Alarm[2] + Contact[2] + Bulb[2] = 1)
                (Alarm[3] + Contact[3] + Bulb[3] = 1)
                (Alarm[4] + Contact[4] + Bulb[4] = 1)
                (Alarm[5] + Contact[5] + Bulb[5] = 1)
                (Alarm[6] + Contact[6] + Bulb[6] = 1)
                (Alarm[7] + Contact[7] + Bulb[7] = 1)
end_concernLevel global_NoE

concernLevel irrigation_NoE
        simpleConstraintState Natural_Irrigation_NoE_state
                Dripline=1
                Emitter=114
end_concernLevel irrigation_NoE

concernLevel heating_NoE
        simpleConstraintState natural_heating_NoE_state
                AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4]=0
        simpleConstraintState automatic_heating_NoE_state
                AirConditionner[1]+AirConditionner[2]+AirConditionner[3]+AirConditionner[4]=1
                Temp[1]+Temp[2]+Temp[3]+Temp[4]=25 /\ Temp[1]*Temp[2]=0 /\ Temp[3]*Temp[4]=0
                /\ Temp[1]*Temp[3]=0 /\ Temp[2]*Temp[4]=0
                Speed[1]+Speed[2]+Speed[3]+Speed[4]=2 /\ Speed[1]*Speed[2]=0 /\
                Speed[3]*Speed[4]=0 /\ Speed[1]*Speed[3]=0 /\ Speed[2]*Speed[4]=0
                Mode[1]+Mode[2]+Mode[3]+Mode[4]=3 /\ Mode[1]*Mode[2]=0 /\ Mode[3]*Mode[4]=0 /\
                Mode[1]*Mode[3]=0 /\ Mode[2]*Mode[4]=0
end_concernLevel heating_NoE

concernLevel moitSensing_NoE
        simpleConstraintState survival_moist_NoE_state
                Master[1]+Master[2]+Master[3]+Master[4]+Master[5]+Master[6]+Master[7]=1
                BatteryLevel[1]<=20 -> Master[1]=0
                BatteryLevel[2]<=20 -> Master[2]=0
```

```
                        BatteryLevel[3]<=20 -> Master[3]=0
                        BatteryLevel[4]<=20 -> Master[4]=0
                        BatteryLevel[5]<=20 -> Master[5]=0
                        BatteryLevel[6]<=20 -> Master[6]=0
                        BatteryLevel[7]<=20 -> Master[7]=0
        end_concernLevel moitSensing_NoE
end_compositeState NoE

//Start transitions
//SS
Start transition to SS.global_SS.global_SS_state
Start transition to SS.irrigation_SS.Natural_Irrigation_SS_state
Start transition to SS.heating_SS.natural_cooling_SS_state
Start transition to SS.moitSensing_SS.acurate_moist_SS_state
Start transition to SS.brightness_SS.natural_brightness_SS_state
Start transition to SS.waterSource_SS.efficientWater_SS_state
////AW
Start transition to AW.global_AW.global_AW_state
Start transition to AW.irrigation_AW.Natural_Irrigation_AW_state
Start transition to AW.heating_AW.normal_heating_AW_state
Start transition to AW.moitSensing_AW.acurate_moist_AW_state
Start transition to AW.brightneAW_AW.natural_brightness_AW_state
Start transition to AW.waterSource_AW.efficientWater_AW_state
////NoE
Start transition to NoE.global_NoE.global_NoE_state
Start transition to NoE.irrigation_NoE.Natural_Irrigation_NoE_state
Start transition to NoE.heating_NoE.natural_heating_NoE_state
Start transition to NoE.moitSensing_NoE.survival_moist_NoE_state

//Composite States
When SS if (PowerFailure=1) transition from SS to NoE
When SS if (Month>=10 \/ Month<4) transition from SS to AW
When SS if (PowerFailure=1) transition from AW to NoE
When SS if (Month>=4 \/ Month<10) transition from AW to SS

//SS Transitions-----------------
When SS  if (Rain=0 \/ IN_close=1)  transition from SS.irrigation_SS.Natural_Irrigation_SS_state to
SS.irrigation_SS.automatic_SS
When  SS  if  (Rain=1  /\  IN_open=1)  transition  from  SS.irrigation_SS.automatic_SS  to
SS.irrigation_SS.Natural_Irrigation_SS_state
//
When SS.irrigation_SS.automatic_SS  if (MinTankWater<TankLevel /\ TankLevel>=OpTankWater) transition
from            SS.irrigation_SS.automatic_SS.automatic_CL_SS.rapidAcc_Irrigation_SS            to
SS.irrigation_SS.automatic_SS.automatic_CL_SS.rapidEff_Irrigation_SS
When    SS.irrigation_SS.automatic_SS       if    (TankLevel<    MinTankWater)    transition    from
SS.irrigation_SS.automatic_SS.automatic_CL_SS.rapidEff_Irrigation_SS                              to
SS.irrigation_SS.automatic_SS.automatic_CL_SS.slowIrrigation_SS
```

```
When SS.irrigation_SS.automatic_SS  if (MinTankWater<TankLevel /\ TankLevel<OpTankWater)  transition
from                SS.irrigation_SS.automatic_SS.automatic_CL_SS.slowIrrigation_SS                to
SS.irrigation_SS.automatic_SS.automatic_CL_SS.rapidEff_Irrigation_SS
When     SS.irrigation_SS.automatic_SS          if     (TankLevel>OpTankWater)     transition     from
SS.irrigation_SS.automatic_SS.automatic_CL_SS.slowIrrigation_SS                                    to
SS.irrigation_SS.automatic_SS.automatic_CL_SS.rapidAcc_Irrigation_SS
When     SS.irrigation_SS.automatic_SS          if     (TankLevel>OpTankWater)     transition     from
SS.irrigation_SS.automatic_SS.automatic_CL_SS.rapidEff_Irrigation_SS                               to
SS.irrigation_SS.automatic_SS.automatic_CL_SS.rapidAcc_Irrigation_SS
//
When  SS     if  (Temperature>30)     transition  from  SS.heating_SS.natural_cooling_SS_state  to
SS.heating_SS.automatic_cooling_SS_state
When  SS     if  (Temperature<=30)     transition  from  SS.heating_SS.automatic_cooling_SS_state  to
SS.heating_SS.natural_cooling_SS_state
When SS  if (BatteryLevel[1]<60 \/ BatteryLevel[2]<60 \/ BatteryLevel[3]<60 \/ BatteryLevel[4]<60 \/
BatteryLevel[5]<60     \/     BatteryLevel[6]<60     \/     BatteryLevel[7]<60)     transition     from
SS.moitSensing_SS.acurate_moist_SS_state to SS.moitSensing_SS.efficient_moit_SS_state
When SS  if (BatteryLevel[1]>= 60 /\ BatteryLevel[2]>= 60 /\ BatteryLevel[3]>= 60 /\ BatteryLevel[4]>=
60  /\  BatteryLevel[5]>=  60  /\  BatteryLevel[6]>=  60  /\  BatteryLevel[7]>=  60)   transition  from
SS.moitSensing_SS.efficient_moit_SS_state to SS.moitSensing_SS.acurate_moist_SS_state
When     SS         if     ((Rain=0     \/     HumidityLevel<NormalHumidity))         transition     from
SS.brightness_SS.automatic_brightness_SS_state to SS.brightness_SS.natural_brightness_SS_state
When     SS         if     (Rain=1     /\     HumidityLevel>NormalHumidity)         transition     from
SS.brightness_SS.natural_brightness_SS_state to SS.brightness_SS.automatic_brightness_SS_state
When SS  if (TankLevel<MinTankWater)  transition from SS.waterSource_SS.efficientWater_SS_state to
SS.waterSource_SS.durableWater_SS_state
When SS  if (TankLevel>=MinTankWater)  transition from SS.waterSource_SS.durableWater_SS_state to
SS.waterSource_SS.efficientWater_SS_state
//
////AW Transitions-----------------
When AW if  (Rain=0 \/ IN_close=1)  transition from AW.irrigation_AW.Natural_Irrigation_AW_state to
AW.irrigation_AW.automatic_AW
When  AW  if     (Rain=1  /\  IN_open=1)       transition  from  AW.irrigation_AW.automatic_AW  to
AW.irrigation_AW.Natural_Irrigation_AW_state
When     AW.irrigation_AW.automatic_AW     if         (TankLevel<OpTankWater)         transition     from
AW.irrigation_AW.automatic_AW.automatic_CL_AW.slowIrrigation_AW                                    to
AW.irrigation_AW.automatic_AW.automatic_CL_AW.rapidIrrigation_AW
When   AW.irrigation_AW.automatic_AW   if      (TankLevel>OpTankWater)             transition   from
AW.irrigation_AW.automatic_AW.automatic_CL_AW.rapidIrrigation_AW                                   to
AW.irrigation_AW.automatic_AW.automatic_CL_AW.slowIrrigation_AW
//
When   AW   if      (IN_close=1   /\   Temperature>5   /\   Temperature<=10)      transition   from
AW.heating_AW.normal_heating_AW_state to AW.heating_AW.high_heating_AW_state
When AW if  (IN_close=1 /\ Temperature>=10)  transition from AW.heating_AW.high_heating_AW_state to
AW.heating_AW.normal_heating_AW_state
When AW  if   (IN_close=1 /\ Temperature>=10)   transition from AW.heating_AW.open_air_AW_state to
AW.heating_AW.normal_heating_AW_state
```

```
When AW if (IN_close=1 /\ Temperature<=5) transition from AW.heating_AW.open_air_AW_state to AW.heating_AW.extra_heating_AW_state
When AW if (IN_open=1) transition from AW.heating_AW.extra_heating_AW_state to AW.heating_AW.open_air_AW_state
When AW if (IN_close=1 /\ Temperature<=5) transition from AW.heating_AW.high_heating_AW_state to AW.heating_AW.extra_heating_AW_state
When AW if (IN_close=1 /\ Temperature>5 /\ Temperature<=10) transition from AW.heating_AW.extra_heating_AW_state to AW.heating_AW.high_heating_AW_state
When AW if (IN_close=1 /\ Temperature>5 /\ Temperature<=10) transition from AW.heating_AW.open_air_AW_state to AW.heating_AW.high_heating_AW_state
//
When AW if (BatteryLevel[1]<50 \/ BatteryLevel[2]<50 \/ BatteryLevel[3]<50 \/ BatteryLevel[4]<50 \/ BatteryLevel[5]<50 \/BatteryLevel[6]<50 \/ BatteryLevel[7]<50) transition from AW.moitSensing_AW.acurate_moist_AW_state to AW.moitSensing_AW.efficient_moit_AW_state
When AW if (BatteryLevel[1]>= 50 /\ BatteryLevel[2]>= 50 /\ BatteryLevel[3]>= 50 /\ BatteryLevel[4]>= 50 /\ BatteryLevel[5]>= 50 /\BatteryLevel[6]>= 50 /\ BatteryLevel[7]>= 50) transition from AW.moitSensing_AW.efficient_moit_AW_state to AW.moitSensing_AW.acurate_moist_AW_state
When AW if ((EConsumption>OpConsumption /\ Brightness>=opBrightness) \/ (EConsumption<OpConsumption /\ Brightness>=minBrightness)) /\ (Rain=0 \/ HumidityLevel<=NormalHumidity) transition from AW.brightneAW_AW.automatic_brightness_AW_state to AW.brightneAW_AW.natural_brightness_AW_state
When AW if (EConsumption>OpConsumption /\ Brightness<opBrightness) \/ (Brightness<minBrightness) \/ (Rain=1 /\ HumidityLevel>NormalHumidity) transition from AW.brightneAW_AW.natural_brightness_AW_state to AW.brightneAW_AW.automatic_brightness_AW_state
When AW if (TankLevel>=MinTankWater) transition from AW.waterSource_AW.durableWater_AW_state to AW.waterSource_AW.efficientWater_AW_state
When AW if (TankLevel<MinTankWater) transition from AW.waterSource_AW.efficientWater_AW_state to AW.waterSource_AW.durableWater_AW_state
//
////NoElectricity Transitions-----------------
When NoE if (Month=10) transition from NoE.heating_NoE.natural_heating_NoE_state to NoE.heating_NoE.automatic_heating_NoE_state
When NoE if (Month=4) transition from NoE.heating_NoE.automatic_heating_NoE_state to NoE.heating_NoE.natural_heating_NoE_state
```

Annexe C : Modèle SCT – Landing Gear System (LGS)

```
stateMachine Landing_Gear_System20


///***************///
///* THE SYSTEM *///
///***************///
Boolean LGS

// PART 1 : Pilot Interface part
//------------------------------
Boolean Pilot_Interface
Boolean Param Enum IN_Handle [0..3]  //0=untouched, 1="up", 2="down", 3=locked

Integer Enum Light[1,3] [0..3] // 0=NotinConf, 1="red", 2="green", 3="orange"

// PART 2 : Mechanic and hydraulic part
//-------------------------------------
Boolean Hydraulic_Circuit
/* Gears */
Boolean Gear_Set [1,3]
Integer Enum f_Gear [0..3]  // 0=NotinConf,  1= "extended", 2= "Retracted", 3= "Interm"
Integer Enum r_Gear [0..3]  // 0=NotinConf,  1= "extended", 2= "Retracted", 3= "Interm"
Integer Enum l_Gear [0..3]  // 0=NotinConf,  1= "extended", 2= "Retracted", 3= "Interm"
Boolean f_G_Latching_box [1,2] [0..2] //  0=NotinConf 2=locked, 1=unlocked
Boolean r_G_Latching_box [1,2] [0..2] //  0=NotinConf 2=locked, 1=unlocked
Boolean l_G_Latching_box [1,2] [0..2] //  0=NotinConf 2=locked, 1=unlocked
/* Doors */
Boolean Door_Set[1,3]
Integer Enum f_Door [0..2] // 0=NotinConf 1="closed", 2="open"
Integer Enum r_Door [0..2] // 0=NotinConf 1="closed", 2="open"
Integer Enum l_Door [0..2] // 0=NotinConf 1="closed", 2="open"
Integer Enum f_D_Latching_box [1,2] [0..2] // 0=NotinConf, 1=unlocked, 0=locked,  // both 1 and 2 are
up
Integer Enum r_D_Latching_box [1,2] [0..2] // 0=NotinConf, 1=unlocked, 0=locked,
Integer Enum l_D_Latching_box [1,2] [0..2] // 0=NotinConf, 1=unlocked, 0=locked,
/* Cylinders *///???????????????
Boolean Cylinder_Set
Integer Enum f_G_Cylinder [0..2] //0=NotinConf, 1=Pressurised, 2=NonPressurized
Integer Enum r_G_Cylinder [0..2] //0=NotinConf, 1=Pressurised, 2=NonPressurized
Integer Enum l_G_Cylinder [0..2] //0=NotinConf, 1=Pressurised, 2=NonPressurized
Integer Enum f_D_Cylinder [0..2] //0=NotinConf, 1=Pressurised, 2=NonPressurized
```

```
Integer Enum r_D_Cylinder [0..2] //0=NotinConf, 1=Pressurised, 2=NonPressurized
Integer Enum l_D_Cylinder [0..2] //0=NotinConf, 1=Pressurised, 2=NonPressurized
Integer Param Enum IN_G_cylinders [0..2] //0=NotinConf, 1=high, 2=down
Integer Param Enum IN_D_cylinders [0..2] //0=NotinConf, 1=high, 2=down
/* Electro-Valves *///???????????????
Boolean EV [1,5]
Boolean Param IN_Hin [1,5]
Boolean Param IN_Hout [1,5]
/* Triplicated Sensors */
Boolean Sensors
Integer Param Enum IN_Handle_Sensor [1,3] [0..2] // 0=NotinConf 1="up", 2="down"
Integer Param Enum IN_f_gearPosition_Sensor [1,3] [0..3]  // 0=NotinConf 1="extended", 2= "Retracted",
3= "Interm"
Integer Param Enum IN_l_gearPosition_Sensor [1,3] [0..3]  // 0=NotinConf 1="extended", 2= "Retracted",
3= "Interm"
Integer Param Enum IN_r_gearPosition_Sensor [1,3] [0..3]  // 0=NotinConf 1="extended", 2= "Retracted",
3= "Interm"
Integer Param Enum IN_f_door_Sensor [1,3] [0..2]  // 0=NotinConf 1="closed", 2= "open"
Integer Param Enum IN_l_door_Sensor [1,3] [0..2]  // 0=NotinConf 1="closed", 2= "open"
Integer Param Enum IN_r_door_Sensor [1,3] [0..2]  // 0=NotinConf 1="closed", 2= "open"

Integer Enum Analog_switch_Sensor [1,3] [0..2] // 0=NotinConf 1="closed", 2="open"
Integer Enum f_shockAbsorber_Sensor [1,3] [0..2] // 0=NotinConf 1="ground", 2= "flight"
Integer Enum l_shockAbsorber_Sensor [1,3] [0..2] // 0=NotinConf 1="ground", 2= "flight"
Integer Enum r_shockAbsorber_Sensor [1,3] [0..2] // 0=NotinConf 1="ground", 2= "flight"
Integer Enum Circuit_pressuried_Sensor [1,3] [0..2] // 0=NotinConf 1="nonPressurized", 2= "pressurized"
// PART 3 : Digital part
//----------------------
/* Duplicated commands */ //???????????????
Boolean Digital_Controller[1,2]
Boolean EV_Commands [1,2]
Boolean Param IN_General_EV[1,2]
Boolean Param IN_close_EV[1,2]
Boolean Param IN_open_EV[1,2]
Boolean Param IN_retract_EV[1,2]
Boolean Param IN_extend_EV[1,2]
Boolean Cockpit_commands [1,2]
Boolean gear_locked_down[1,2]
Boolean gear_maneuvering [1,2]
Boolean Param IN_anomaly [1,2]
//PART4 : Emergency Part
//----------------------
Boolean Manual_Crank
Boolean Analog_Controller
//Time related variables
//----------------------
Boolean Param IN_Time_to_close_switch
```

```
Float Param AS_Time_to_open
Float Param AS_Time_to_open_limit
Float Param AS_Time_to_close
Float Param AS_Time_to_close_limit
Float Param EV_Time_to_open
Float Param EV_Time_to_open_limit
Float Param EV_Time_to_close
Float Param EV_Time_to_close_limit
Float Param Cy_Time_to_down
Float Param Cy_Time_to_down_limit
Float Param Cy_Time_to_high
Float Param Cy_Time_to_high_limit
Boolean random
//OUT variables
//-------------
Integer Enum OUT_Handle_Sensor [1,3] [0..2] // 0=NotinConf 1="up", 2="down"
Boolean Enum OUT_Handle [0..3]  //0=untouched, 1="up", 2="down", 3=locked
Integer Enum OUT_G_cylinders [0..2] //0=NotinConf, 1=high, 2=down
Integer Enum OUT_D_cylinders [0..2] //0=NotinConf, 1=high, 2=down
Boolean OUT_Hin [1,5]
Boolean OUT_Hout [1,5]
Integer Enum OUT_f_gearPosition_Sensor [1,3] [0..3]  // 0=NotinConf 1="extended", 2= "Retracted", 3=
"Interm"
Integer Enum OUT_l_gearPosition_Sensor [1,3] [0..3]  // 0=NotinConf 1="extended", 2= "Retracted", 3=
"Interm"
Integer Enum OUT_r_gearPosition_Sensor [1,3] [0..3]  // 0=NotinConf 1="extended", 2= "Retracted", 3=
"Interm"
Integer Enum OUT_f_door_Sensor [1,3] [0..2]  // 0=NotinConf 1="closed", 2= "open"
Integer Enum OUT_l_door_Sensor [1,3] [0..2]  // 0=NotinConf 1="closed", 2= "open"
Integer Enum OUT_r_door_Sensor [1,3] [0..2]  // 0=NotinConf 1="closed", 2= "open"
Boolean OUT_General_EV[1,2]
Boolean OUT_close_EV[1,2]
Boolean OUT_open_EV[1,2]
Boolean OUT_retract_EV[1,2]
Boolean OUT_extend_EV[1,2]
Boolean OUT_anomaly [1,2]
Boolean OUT_Time_to_close_switch

///************************///
///* FLEXIBLE CONSTRAINTS *///
///************************///
LGS=1
LGS=Pilot_Interface
LGS>=Hydraulic_Circuit
2*LGS>=Digital_Controller[1]+Digital_Controller[2]
Hydraulic_Circuit=1 -> Digital_Controller[1]+Digital_Controller[2]=2
Hydraulic_Circuit*Analog_Controller=0
```

```
//Part1:
Pilot_Interface=1
Pilot_Interface>0 -> OUT_Handle >0 /\ (Light[1]>1 /\ Light[2]>1 /\ Light[3]>1)
//Part2:
Hydraulic_Circuit = Cylinder_Set
f_D_Cylinder >0 <-> Cylinder_Set>0
r_D_Cylinder >0 <-> Cylinder_Set>0
l_D_Cylinder >0 <-> Cylinder_Set>0
f_G_Cylinder >0 <-> Cylinder_Set>0
r_G_Cylinder >0 <-> Cylinder_Set>0
l_G_Cylinder >0 <-> Cylinder_Set>0
//--------------------------------
Hydraulic_Circuit=1 -> (Gear_Set[1]+Gear_Set[2]+Gear_Set[3] = 3 )
f_Gear>0 <-> Gear_Set[1]=1
(f_G_Latching_box[1]*f_G_Latching_box[2]>0) <-> Gear_Set[1]=1
r_Gear>0 <-> Gear_Set[2]=1
(r_G_Latching_box[1]*r_G_Latching_box[2]>0) <-> Gear_Set[2]=1
l_Gear>0 <-> Gear_Set[3]=1
(l_G_Latching_box[1]*l_G_Latching_box[2]>0) <-> Gear_Set[3]=1
//-----------------------------------------------------------------
Hydraulic_Circuit=1 -> (Door_Set[1]+Door_Set[2]+Door_Set[3] = 3 )
f_Door>0 <-> Door_Set[1]=1
(f_D_Latching_box[1]*f_D_Latching_box[2]>0) <-> Door_Set[1]=1
r_Door>0 <-> Door_Set[2]=1
(r_D_Latching_box[1]*r_D_Latching_box[2]>0) <-> Door_Set[2]=1
l_Door>0 <-> Door_Set[3]=1
(l_D_Latching_box[1]*l_D_Latching_box[2]>0) <-> Door_Set[3]=1
//-------------------------------------------------------------
Hydraulic_Circuit=Sensors
Analog_switch_Sensor[1]*Analog_switch_Sensor[2]*Analog_switch_Sensor[3] >0 <-> Sensors>0
OUT_f_gearPosition_Sensor[1]*OUT_f_gearPosition_Sensor[2]*OUT_f_gearPosition_Sensor[3]     >0     <->
Sensors>0
OUT_r_gearPosition_Sensor[1]*OUT_r_gearPosition_Sensor[2]*OUT_r_gearPosition_Sensor[3]     >0     <->
Sensors>0
OUT_l_gearPosition_Sensor[1]*OUT_l_gearPosition_Sensor[2]*OUT_l_gearPosition_Sensor[3]     >0     <->
Sensors>0
OUT_f_door_Sensor[1]*OUT_f_door_Sensor[2]*OUT_f_door_Sensor[3] >0 <-> Sensors>0
OUT_r_door_Sensor[1]*OUT_r_door_Sensor[2]*OUT_r_door_Sensor[3] >0 <-> Sensors>0
OUT_l_door_Sensor[1]*OUT_l_door_Sensor[2]*OUT_l_door_Sensor[3] >0 <-> Sensors>0
f_shockAbsorber_Sensor[1]*f_shockAbsorber_Sensor[2]*f_shockAbsorber_Sensor[3] >0 <-> Sensors>0
r_shockAbsorber_Sensor[1]*r_shockAbsorber_Sensor[2]*r_shockAbsorber_Sensor[3] >0 <-> Sensors>0
l_shockAbsorber_Sensor[1]*l_shockAbsorber_Sensor[2]*l_shockAbsorber_Sensor[3] >0 <-> Sensors>0
Circuit_pressuried_Sensor[1]*Circuit_pressuried_Sensor[2]*Circuit_pressuried_Sensor[3]     >0     <->
Sensors>0
//-----------------------------------------------------
Hydraulic_Circuit>0 <-> EV[1]*EV[2]*EV[3]*EV[4]*EV[5]>0
OUT_Hin[1]>0 <-> EV[1]>0
```

```
OUT_Hin[2]>0 <-> EV[2]>0
OUT_Hin[3]>0 <-> EV[3]>0
OUT_Hin[4]>0 <-> EV[4]>0
OUT_Hin[5]>0 <-> EV[5]>0
OUT_Hout[1]>0 -> EV[1]>0
OUT_Hout[2]>0 -> EV[2]>0
OUT_Hout[3]>0 -> EV[3]>0
OUT_Hout[4]>0 -> EV[4]>0
OUT_Hout[5]>0 -> EV[5]>0
//Part3
EV_Commands[1]=Digital_Controller[1]
EV_Commands[1]>=OUT_retract_EV[1]           /\           EV_Commands[1]>=OUT_extend_EV[1]           /\
OUT_retract_EV[1]*OUT_extend_EV[1]=0
EV_Commands[1]>=OUT_open_EV[1] /\ EV_Commands[1]>=OUT_close_EV[1] /\ OUT_open_EV[1]*OUT_close_EV[1]=0
OUT_General_EV[1] >0 ->  EV_Commands[1]>0
//-----------------------------------
EV_Commands[2]=Digital_Controller[2]
EV_Commands[2]>=OUT_retract_EV[2]           /\           EV_Commands[2]>=OUT_extend_EV[2]           /\
OUT_retract_EV[2]*OUT_extend_EV[2]=0
EV_Commands[2]>=OUT_open_EV[2] /\ EV_Commands[2]>=OUT_close_EV[2] /\ OUT_open_EV[2]*OUT_close_EV[2]=0
OUT_General_EV[2] >0 ->  EV_Commands[2]>0
//--------------------------------------
Cockpit_commands[1]=Digital_Controller[1]
Cockpit_commands[1] >= gear_locked_down[1]
Cockpit_commands[1] >= gear_maneuvering[1]
Cockpit_commands[1] >= OUT_anomaly[1]
gear_locked_down[1] + gear_maneuvering[1] + OUT_anomaly[1] <= 3
//--------------------------------------
Cockpit_commands[2]=Digital_Controller[2]
Cockpit_commands[2] >= gear_locked_down[2]
Cockpit_commands[2] >= gear_maneuvering[2]
Cockpit_commands[2] >= OUT_anomaly[2]
gear_locked_down[2] + gear_maneuvering[2] + OUT_anomaly[2] <= 3
//Part4
LGS>=Manual_Crank
LGS>=Analog_Controller

//GEneric constraints
OUT_Time_to_close_switch=0
OUT_G_cylinders = IN_G_cylinders
OUT_D_cylinders = IN_D_cylinders
OUT_Handle>0

//General constraints
compositeState LGS_Nominal
        concernLevel general_Nominal_cl
                simpleConstraintState general_Nominal
```

```
                Hydraulic_Circuit=1
                Digital_Controller [1]+Digital_Controller [2]=2
end_concernLevel general_Nominal_cl

concernLevel handle
        simpleConstraintState handle_up
                OUT_Handle_Sensor [ 1 ]=1 /\ OUT_Handle_Sensor [ 2 ]=1 /\ OUT_Handle_Sensor [
                3]=1
        simpleConstraintState handle_down
                OUT_Handle_Sensor [ 1 ]=2 /\ OUT_Handle_Sensor [ 2 ]=2 /\ OUT_Handle_Sensor [
                3 ]=2
        simpleConstraintState handle_inactive
                OUT_Handle_Sensor [ 1 ]=0 /\ OUT_Handle_Sensor [ 2 ]=0 /\  OUT_Handle_Sensor
                [ 3 ]=0 /\ OUT_General_EV[1] =0 /\ OUT_General_EV[2] =0
end_concernLevel handle

concernLevel gears_cl
        simpleConstraintState retracted
                OUT_f_gearPosition_Sensor [1] = 2
                OUT_f_gearPosition_Sensor [2] = 2
                OUT_f_gearPosition_Sensor [3] = 2
                OUT_r_gearPosition_Sensor [1] = 2
                OUT_r_gearPosition_Sensor [2] = 2
                OUT_r_gearPosition_Sensor [3] = 2
                OUT_l_gearPosition_Sensor [1] = 2
                OUT_l_gearPosition_Sensor [2] = 2
                OUT_l_gearPosition_Sensor [3] = 2
                Light[1]=0
                Light[2]=0
                Light[3]=0
                (IN_D_cylinders=2   /\   (   (IN_f_gearPosition_Sensor   [1]   =   2   \/
                IN_f_gearPosition_Sensor [2] = 2 \/ IN_f_gearPosition_Sensor [3] = 2 \/
                IN_r_gearPosition_Sensor [1] = 2  \/ IN_r_gearPosition_Sensor [2] = 2 \/
                IN_r_gearPosition_Sensor [3] = 2 \/ IN_l_gearPosition_Sensor [1] = 2 \/
                IN_l_gearPosition_Sensor [2] = 2 \/ IN_l_gearPosition_Sensor [3] = 2)) /\
                (IN_Handle_Sensor[1]=1 /\ IN_Handle_Sensor[2]=1 /\ IN_Handle_Sensor[3]=1)) ->
                OUT_retract_EV[1]=0 /\ OUT_retract_EV[2]=0
                (IN_D_cylinders=2   /\   (   (IN_f_gearPosition_Sensor   [1]   =   2   \/
                IN_f_gearPosition_Sensor [2] = 2 \/ IN_f_gearPosition_Sensor [3] = 2 \/
                IN_r_gearPosition_Sensor [1] = 2  \/ IN_r_gearPosition_Sensor [2] = 2 \/
                IN_r_gearPosition_Sensor [3] = 2 \/ IN_l_gearPosition_Sensor [1] = 2 \/
                IN_l_gearPosition_Sensor [2] = 2 \/ IN_l_gearPosition_Sensor [3] = 2)) /\
                (IN_Handle_Sensor[1]=1 /\ IN_Handle_Sensor[2]=1 /\ IN_Handle_Sensor[3]=1)) ->
                OUT_open_EV[1]=0 /\ OUT_open_EV[2]=0
                (IN_D_cylinders=2   /\   (   (IN_f_gearPosition_Sensor   [1]   =   2   \/
                IN_f_gearPosition_Sensor [2] = 2 \/ IN_f_gearPosition_Sensor [3] = 2 \/
                IN_r_gearPosition_Sensor [1] = 2  \/ IN_r_gearPosition_Sensor [2] = 2 \/
```

```
        IN_r_gearPosition_Sensor [3] = 2 \/ IN_l_gearPosition_Sensor [1] = 2 \/
        IN_l_gearPosition_Sensor [2] = 2 \/ IN_l_gearPosition_Sensor [3] = 2)) /\
        (IN_Handle_Sensor[1]=1 /\ IN_Handle_Sensor[2]=1 /\ IN_Handle_Sensor[3]=1)) ->
        (OUT_close_EV[1]=1 /\ OUT_close_EV[2]=1)
        ((IN_Hout[1] = IN_Hin[1] /\ IN_Hout[2] = 0 /\ IN_Hout[3] = 0 /\  IN_Hout[4] =
        0  /\  IN_Hout[5]  =  0  )  /\  (IN_f_gearPosition_Sensor  [1]  =  2  \/
        IN_f_gearPosition_Sensor [2] = 2\/ IN_f_gearPosition_Sensor [3] = 2 \/
        IN_r_gearPosition_Sensor [1] = 2  \/ IN_r_gearPosition_Sensor [2] = 2 \/
        IN_r_gearPosition_Sensor [3] = 2 \/ IN_l_gearPosition_Sensor [1] = 2 \/
        IN_l_gearPosition_Sensor [2] = 2 \/ IN_l_gearPosition_Sensor [3] = 2) /\
        (IN_Handle_Sensor[1]=1 /\ IN_Handle_Sensor[2]=1)) -> (OUT_Handle=0)
simpleConstraintState extended
        OUT_f_gearPosition_Sensor [1] = 1
        OUT_f_gearPosition_Sensor [2] = 1
        OUT_f_gearPosition_Sensor [3] = 1
        OUT_r_gearPosition_Sensor [1] = 1
        OUT_r_gearPosition_Sensor [2] = 1
        OUT_r_gearPosition_Sensor [3] = 1
        OUT_l_gearPosition_Sensor [1] = 1
        OUT_l_gearPosition_Sensor [2] = 1
        OUT_l_gearPosition_Sensor [3] = 1
        Light[1]=2
        Light[2]=2
        Light[3]=2
        (IN_D_cylinders=2   /\      ((IN_f_gearPosition_Sensor   [1]   =   1   \/
        IN_f_gearPosition_Sensor [2] = 1 \/ IN_f_gearPosition_Sensor [3] = 1 \/
        IN_r_gearPosition_Sensor [1] = 1  \/ IN_r_gearPosition_Sensor [2] = 1 \/
        IN_r_gearPosition_Sensor [3] = 1 \/ IN_l_gearPosition_Sensor [1] = 1 \/
        IN_l_gearPosition_Sensor [2] = 1 \/ IN_l_gearPosition_Sensor [3] = 1)) /\
        (IN_Handle_Sensor[1]=2 /\ IN_Handle_Sensor[2]=2 /\ IN_Handle_Sensor[3]=2)) ->
        OUT_extend_EV[1]=0 /\ OUT_extend_EV[2]=0
        (IN_D_cylinders=2   /\      ((IN_f_gearPosition_Sensor   [1]   =   1   \/
        IN_f_gearPosition_Sensor [2] = 1 \/ IN_f_gearPosition_Sensor [3] = 1 \/
        IN_r_gearPosition_Sensor [1] = 1  \/ IN_r_gearPosition_Sensor [2] = 1 \/
        IN_r_gearPosition_Sensor [3] = 1 \/ IN_l_gearPosition_Sensor [1] = 1 \/
        IN_l_gearPosition_Sensor [2] = 1 \/ IN_l_gearPosition_Sensor [3] = 1)) /\
        (IN_Handle_Sensor[1]=2 /\ IN_Handle_Sensor[2]=2 /\ IN_Handle_Sensor[3]=2)) ->
        (OUT_open_EV[1]=0 /\ OUT_open_EV[2]=0)
        (IN_D_cylinders=2   /\      ((IN_f_gearPosition_Sensor   [1]   =   1   \/
        IN_f_gearPosition_Sensor [2] = 1 \/ IN_f_gearPosition_Sensor [3] = 1 \/
        IN_r_gearPosition_Sensor [1] = 1  \/ IN_r_gearPosition_Sensor [2] = 1 \/
        IN_r_gearPosition_Sensor [3] = 1 \/ IN_l_gearPosition_Sensor [1] = 1 \/
        IN_l_gearPosition_Sensor [2] = 1 \/ IN_l_gearPosition_Sensor [3] = 1)) /\
        (IN_Handle_Sensor[1]=2 /\ IN_Handle_Sensor[2]=2 /\ IN_Handle_Sensor[3]=2)) ->
        (OUT_close_EV[1]=1 /\ OUT_close_EV[2]=1)
        gear_locked_down [1] =1 /\ gear_locked_down [2] =1
```

```
                ((IN_Hout[1] = IN_Hin[1] /\ IN_Hout[2] = 0 /\ IN_Hout[3] = 0 /\ IN_Hout[4] =
                0 /\ IN_Hout[5] = 0) /\ (IN_f_gearPosition_Sensor [1] = 1 \/
                IN_f_gearPosition_Sensor [2] = 1 \/ IN_f_gearPosition_Sensor [3] = 1 \/
                IN_r_gearPosition_Sensor [1] = 1 \/ IN_r_gearPosition_Sensor [2] = 1 \/
                IN_r_gearPosition_Sensor [3] = 1 \/ IN_l_gearPosition_Sensor [1] = 1 \/
                IN_l_gearPosition_Sensor [2] = 1 \/ IN_l_gearPosition_Sensor [3] = 1) /\
                (IN_Handle_Sensor[1]=2 /\ IN_Handle_Sensor[2]=2)) -> (OUT_Handle=0)
        simpleConstraintState failure_gear
                OUT_anomaly[1]=1 /\ OUT_anomaly[2]=1
end_concernLevel gears_cl

concernLevel doors_cl
        simpleConstraintState Opened
                OUT_f_door_Sensor [1] = 2
                OUT_f_door_Sensor [2] = 2
                OUT_f_door_Sensor [3] = 2
                OUT_r_door_Sensor [1] = 2
                OUT_r_door_Sensor [2] = 2
                OUT_r_door_Sensor [3] = 2
                OUT_l_door_Sensor [1] = 2
                OUT_l_door_Sensor [2] = 2
                OUT_l_door_Sensor [3] = 2
                (IN_Hout[3] = 1) /\ (IN_D_cylinders=2) /\ (IN_f_gearPosition_Sensor [1] = 2
                \/ IN_f_gearPosition_Sensor [2] = 2 \/ IN_f_gearPosition_Sensor [3] = 2 \/
                IN_r_gearPosition_Sensor [1] = 2 \/ IN_r_gearPosition_Sensor [2] = 2 \/
                IN_r_gearPosition_Sensor [3] = 2 \/ IN_l_gearPosition_Sensor [1] = 2 \/
                IN_l_gearPosition_Sensor [2] = 2 \/ IN_l_gearPosition_Sensor [3] = 2) /\
                (IN_Handle_Sensor[1]=2 /\ IN_Handle_Sensor[2]=2)) -> (OUT_extend_EV[1]=1 /\
                OUT_extend_EV[2]=1)
                (IN_Hout[3] = 1) /\ (IN_D_cylinders=2) /\ (IN_f_gearPosition_Sensor [1] = 1 \/
                IN_f_gearPosition_Sensor [2] = 1 \/ IN_f_gearPosition_Sensor [3] = 1 \/
                IN_r_gearPosition_Sensor [1] = 1 \/ IN_r_gearPosition_Sensor [2] = 1 \/
                IN_r_gearPosition_Sensor [3] = 1 \/ IN_l_gearPosition_Sensor [1] = 1 \/
                IN_l_gearPosition_Sensor [2] = 1 \/ IN_l_gearPosition_Sensor [3] = 1) /\
                (IN_Handle_Sensor[1]=1 /\ IN_Handle_Sensor[2]=1)) -> (OUT_retract_EV[1]=1 /\
                OUT_retract_EV[2]=1)
        simpleConstraintState Closed
                OUT_f_door_Sensor [1] = 1
                OUT_f_door_Sensor [2] = 1
                OUT_f_door_Sensor [3] = 1
                OUT_r_door_Sensor [1] = 1
                OUT_r_door_Sensor [2] = 1
                OUT_r_door_Sensor [3] = 1
                OUT_l_door_Sensor [1] = 1
                OUT_l_door_Sensor [2] = 1
                OUT_l_door_Sensor [3] = 1
        simpleConstraintState failure_doors
```

```
                OUT_anomaly[1]=1 /\ OUT_anomaly[2]=1
end_concernLevel doors_cl

concernLevel cynlinders_cl
        compositeState cylinder_states
                concernLevel gear_cylinder_cl
                        simpleConstraintState inactive_G_cylinders
                                random=1
                        simpleConstraintState G_cylinders_high
                                f_G_Latching_box[1]= 2
                                f_G_Latching_box[2]= 1
                                r_G_Latching_box[1]= 2
                                r_G_Latching_box[2]= 1
                                l_G_Latching_box[1]= 2
                                l_G_Latching_box[2]= 1
                                OUT_G_cylinders=1
                        simpleConstraintState G_cylinders_down
                                f_G_Latching_box[1]= 1
                                f_G_Latching_box[2]= 2
                                r_G_Latching_box[1]= 1
                                r_G_Latching_box[2]= 2
                                l_G_Latching_box[1]= 1
                                l_G_Latching_box[2]= 2
                                OUT_G_cylinders=2
                end_concernLevel gear_cylinder_cl

                concernLevel door_cylinder_cl
                        simpleConstraintState inactive_D_cylinders
                                random=1
                        simpleConstraintState D_cylinders_high
                                f_D_Latching_box[1]= 2
                                f_D_Latching_box[2]= 2
                                r_D_Latching_box[1]= 2
                                r_D_Latching_box[2]= 2
                                l_D_Latching_box[1]= 2
                                l_D_Latching_box[2]= 2
                                OUT_D_cylinders=1
                        simpleConstraintState D_cylinders_down
                                OUT_D_cylinders=2
                end_concernLevel door_cylinder_cl
        end_compositeState cylinder_states

        simpleConstraintState failure_cylinder
                OUT_anomaly[1]=1 /\ OUT_anomaly[2]=1
        end_concernLevel cynlinders_cl

concernLevel electro_valves_cl
```

```
compositeState electro_valves_states
        concernLevel general_EV_cl
                simpleConstraintState nn_pressurized_general_EV
                        OUT_Hout[1] = 0
                simpleConstraintState pressurized_general_EV
                        OUT_Hout[1] = OUT_Hin[1]
                        Circuit_pressuried_Sensor[1]=2
                        Circuit_pressuried_Sensor[2]=2
                        Circuit_pressuried_Sensor[3]=2
                        ((IN_Hout [5]=0  /\ IN_extend_EV [ 1 ]=0 /\ IN_extend_EV [ 2
                        ]=0   )   /\   (IN_f_gearPosition_Sensor   [1]   =   1   /\
                        IN_Handle_Sensor[1]=1) /\ (IN_f_gearPosition_Sensor [1] = 1 /\
                        IN_Handle_Sensor[1]=1)) -> OUT_open_EV [1]=1 /\ OUT_open_EV
                        [2]=1
                        ((IN_Hout [4]=0  /\ IN_retract_EV [ 1 ]=0 /\ IN_retract_EV [ 2
                        ]=0   )   /\   (IN_f_gearPosition_Sensor   [1]   =   2   /\
                        IN_Handle_Sensor[1]=2) /\ (IN_f_gearPosition_Sensor [1] = 2 /\
                        IN_Handle_Sensor[1]=2)) -> OUT_open_EV [1]=1 /\ OUT_open_EV
                        [2]=1
        end_concernLevel general_EV_cl

        concernLevel close_EV_cl
                simpleConstraintState nn_pressurized_close_EV
                        /*A*/OUT_Hout[2] = 0
                simpleConstraintState pressurized_close_EV
                        OUT_Hout[2] = OUT_Hin[2]
        end_concernLevel close_EV_cl

        concernLevel open_EV_cl
                simpleConstraintState nn_pressurized_open_EV
                        /*A*/OUT_Hout[3] = 0
                simpleConstraintState pressurized_open_EV
                        OUT_Hout[3] = OUT_Hin[3]
                end_concernLevel open_EV_cl

        concernLevel retract_EV_cl
                simpleConstraintState nn_pressurized_retract_EV
                        /*A*/OUT_Hout[4] = 0
                simpleConstraintState pressurized_retract_EV
                        OUT_Hout[4] = OUT_Hin[4]
                end_concernLevel retract_EV_cl

        concernLevel extend_EV_cl
                simpleConstraintState nn_pressurized_extend_EV
                        OUT_Hout[5] = 0
                simpleConstraintState pressurized_extend_EV
                        OUT_Hout[5] = OUT_Hin[5]
```

```
                        end_concernLevel extend_EV_cl
                end_compositeState electro_valves_states
                simpleConstraintState failure_EV
                        OUT_anomaly[1]=1 /\ OUT_anomaly[2]=1
        end_concernLevel electro_valves_cl

        concernLevel Analog_switch_cl
                simpleConstraintState Analog_switch_closed
                        OUT_General_EV[1] =1
                        OUT_General_EV[2] =1
                        Analog_switch_Sensor [1] = 2
                        Analog_switch_Sensor [2] = 2
                        Analog_switch_Sensor [3] = 2
                        OUT_Time_to_close_switch=1
                simpleConstraintState Analog_switch_open
                        Analog_switch_Sensor [1] = 1
                        Analog_switch_Sensor [2] = 1
                        Analog_switch_Sensor [3] = 1
                        (IN_Handle_Sensor [ 1 ]>0 /\ IN_Handle_Sensor [ 2 ]>0 /\ IN_Handle_Sensor [ 3
                        ]>0) -> OUT_Time_to_close_switch=1
                        ((IN_Handle>0) /\ IN_Time_to_close_switch=1)   -> (OUT_General_EV[1] =1 /\
                        OUT_General_EV[2] =1)
                simpleConstraintState failure_analog_switch
                        OUT_anomaly[1]=1 /\ OUT_anomaly[2]=1
        end_concernLevel Analog_switch_cl
end_compositeState LGS_Nominal

compositeState LGS_Analog
        concernLevel general_Alanlog_cl
                simpleConstraintState general_Analog
                        Analog_Controller=1
        end_concernLevel general_Alanlog_cl
end_compositeState LGS_Analog

Start transition to LGS_Nominal
Start transition to LGS_Nominal.handle.handle_up
Start transition to LGS_Analog.general_Alanlog_cl.general_Analog
Start transition to LGS_Nominal.gears_cl.retracted
Start transition to LGS_Nominal.doors_cl.Closed
Start transition to LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.G_cylinders_high
Start transition to LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.D_cylinders_high
Start                                        transition                                        to
LGS_Nominal.electro_valves_cl.electro_valves_states.close_EV_cl.pressurized_close_EV
Start                                        transition                                        to
LGS_Nominal.electro_valves_cl.electro_valves_states.extend_EV_cl.nn_pressurized_extend_EV
Start                                        transition                                        to
LGS_Nominal.electro_valves_cl.electro_valves_states.general_EV_cl.nn_pressurized_general_EV
```

```
Start transition to LGS_Nominal.electro_valves_cl.electro_valves_states.open_EV_cl.nn_pressurized_open_EV
Start transition to LGS_Nominal.electro_valves_cl.electro_valves_states.retract_EV_cl.pressurized_retract_EV
Start transition to LGS_Nominal.Analog_switch_cl.Analog_switch_open
Start transition to LGS_Nominal.cynlinders_cl.cylinder_states
//Transitions on the handle
When LGS_Nominal if IN_Handle=1 transition from LGS_Nominal.handle.handle_down to LGS_Nominal.handle.handle_up
When LGS_Nominal if IN_Handle=2 transition from LGS_Nominal.handle.handle_up to LGS_Nominal.handle.handle_down
When LGS_Nominal if IN_Handle=0 transition from LGS_Nominal.handle.handle_down to LGS_Nominal.handle.handle_inactive
When LGS_Nominal if IN_Handle=0 transition from LGS_Nominal.handle.handle_up to LGS_Nominal.handle.handle_inactive
//Transition on gears states
When LGS_Nominal if IN_G_cylinders=1 transition from LGS_Nominal.gears_cl.extended to LGS_Nominal.gears_cl.retracted
When LGS_Nominal if IN_G_cylinders=2 transition from LGS_Nominal.gears_cl.retracted to LGS_Nominal.gears_cl.extended
////Transition on doors states
When LGS_Nominal if IN_D_cylinders=1 transition from LGS_Nominal.doors_cl.Opened to LGS_Nominal.doors_cl.Closed
When LGS_Nominal if IN_D_cylinders=2 transition from LGS_Nominal.doors_cl.Closed to LGS_Nominal.doors_cl.Opened
//Transition on first level cylinders states
When LGS_Nominal if (Cy_Time_to_down<Cy_Time_to_down_limit) transition from LGS_Nominal.cynlinders_cl.failure_cylinder to LGS_Nominal.cynlinders_cl.cylinder_state
When LGS_Nominal if (Cy_Time_to_down>Cy_Time_to_down_limit) transition from LGS_Nominal.cynlinders_cl.cylinder_states to LGS_Nominal.cynlinders_cl.failure_cylinder
//----------
When LGS_Nominal.cynlinders_cl.cylinder_states if (IN_Hout[4] = 0 /\ IN_Hout[5] = 0) transition from LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.G_cylinders_down to LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.inactive_G_cylinders
When LGS_Nominal.cynlinders_cl.cylinder_states if (IN_Hout[4] = 0 /\ IN_Hout[5] = 0) transition from LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.G_cylinders_high to LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.inactive_G_cylinders
When LGS_Nominal.cynlinders_cl.cylinder_states if (IN_Hout[2] = 0 /\ IN_Hout[3] = 0) transition from LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.D_cylinders_high to LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.inactive_D_cylinders
When LGS_Nominal.cynlinders_cl.cylinder_states if (IN_Hout[2] = 0 /\ IN_Hout[3] = 0) transition from LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.D_cylinders_down to LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.inactive_D_cylinders
When LGS_Nominal.cynlinders_cl.cylinder_states if IN_Hout[4] = IN_Hin[4] transition from LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.G_cylinders_down to LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.G_cylinders_high
```

```
When LGS_Nominal.cynlinders_cl.cylinder_states if IN_Hout[5] = IN_Hin[5] transition from
LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.G_cylinders_high to
LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.G_cylinders_down
When LGS_Nominal.cynlinders_cl.cylinder_states if IN_Hout[2] = IN_Hin[2] transition from
LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.D_cylinders_down to
LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.D_cylinders_high
When LGS_Nominal.cynlinders_cl.cylinder_states if IN_Hout[3] = IN_Hin[3] transition from
LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.D_cylinders_high to
LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.D_cylinders_down
When LGS_Nominal.cynlinders_cl.cylinder_states if IN_Hout[4] = IN_Hin[4] transition from
LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.inactive_G_cylinders to
LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.G_cylinders_high
When LGS_Nominal.cynlinders_cl.cylinder_states if IN_Hout[5] = IN_Hin[5] transition from
LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.inactive_G_cylinders to
LGS_Nominal.cynlinders_cl.cylinder_states.gear_cylinder_cl.G_cylinders_down
When LGS_Nominal.cynlinders_cl.cylinder_states if IN_Hout[2] = IN_Hin[2] transition from
LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.inactive_D_cylinders to
LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.D_cylinders_high
When LGS_Nominal.cynlinders_cl.cylinder_states if IN_Hout[3] = IN_Hin[3] transition from
LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.inactive_D_cylinders to
LGS_Nominal.cynlinders_cl.cylinder_states.door_cylinder_cl.D_cylinders_down
//Transition on Electro_vale states
When LGS_Nominal if ((EV_Time_to_open<EV_Time_to_open_limit) /\
(EV_Time_to_close<EV_Time_to_close_limit)) transition from LGS_Nominal.electro_valves_cl.failure_EV
to LGS_Nominal.electro_valves_cl.electro_valves_states
When LGS_Nominal if ((EV_Time_to_open>EV_Time_to_open_limit) \/
(EV_Time_to_close>EV_Time_to_close_limit)) transition from
LGS_Nominal.electro_valves_cl.electro_valves_states to LGS_Nominal.electro_valves_cl.failure_EV
//----------
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_General_EV[1] =0 \/ IN_General_EV[2]
=0) transition from
LGS_Nominal.electro_valves_cl.electro_valves_states.general_EV_cl.pressurized_general_EV to
LGS_Nominal.electro_valves_cl.electro_valves_states.general_EV_cl.nn_pressurized_general_EV
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_open_EV[1] =0 \/ IN_open_EV[2] =0)
transition from LGS_Nominal.electro_valves_cl.electro_valves_states.open_EV_cl.pressurized_open_EV to
LGS_Nominal.electro_valves_cl.electro_valves_states.open_EV_cl.nn_pressurized_open_EV
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_close_EV[1] =0 \/ IN_close_EV[2] =0)
transition from LGS_Nominal.electro_valves_cl.electro_valves_states.close_EV_cl.pressurized_close_EV
to LGS_Nominal.electro_valves_cl.electro_valves_states.close_EV_cl.nn_pressurized_close_EV
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_retract_EV[1] =0 \/ IN_retract_EV[2]
=0) transition from
LGS_Nominal.electro_valves_cl.electro_valves_states.retract_EV_cl.pressurized_retract_EV to
LGS_Nominal.electro_valves_cl.electro_valves_states.retract_EV_cl.nn_pressurized_retract_EV
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_extend_EV[1] =0 \/ IN_extend_EV[2] =0)
transition from
LGS_Nominal.electro_valves_cl.electro_valves_states.extend_EV_cl.pressurized_extend_EV to
LGS_Nominal.electro_valves_cl.electro_valves_states.extend_EV_cl.nn_pressurized_extend_EV
```

```
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_General_EV[1] =1 \/ IN_General_EV[2]
=1) transition from
LGS_Nominal.electro_valves_cl.electro_valves_states.general_EV_cl.nn_pressurized_general_EV to
LGS_Nominal.electro_valves_cl.electro_valves_states.general_EV_cl.pressurized_general_EV
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_open_EV[1] =1 \/ IN_open_EV[2] =1)
transition from LGS_Nominal.electro_valves_cl.electro_valves_states.open_EV_cl.nn_pressurized_open_EV
to LGS_Nominal.electro_valves_cl.electro_valves_states.open_EV_cl.pressurized_open_EV
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_close_EV[1] =1 \/ IN_close_EV[2] =1)
transition from
LGS_Nominal.electro_valves_cl.electro_valves_states.close_EV_cl.nn_pressurized_close_EV to
LGS_Nominal.electro_valves_cl.electro_valves_states.close_EV_cl.pressurized_close_EV
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_retract_EV[1] =1 \/ IN_retract_EV[2]
=1) transition from
LGS_Nominal.electro_valves_cl.electro_valves_states.retract_EV_cl.nn_pressurized_retract_EV to
LGS_Nominal.electro_valves_cl.electro_valves_states.retract_EV_cl.pressurized_retract_EV
When LGS_Nominal.electro_valves_cl.electro_valves_states if (IN_extend_EV[1] =1 \/ IN_extend_EV[2] =1)
transition from
LGS_Nominal.electro_valves_cl.electro_valves_states.extend_EV_cl.nn_pressurized_extend_EV to
LGS_Nominal.electro_valves_cl.electro_valves_states.extend_EV_cl.pressurized_extend_EV
//Transition of Analog Switch
/*A*/When LGS_Nominal if (IN_Time_to_close_switch=0 /\ ((IN_Handle_Sensor [ 1 ]=1 /\ IN_Handle_Sensor
[ 2 ]=1 /\ IN_Handle_Sensor [ 3 ]=1) \/ (IN_Handle_Sensor [ 1 ]=2 /\ IN_Handle_Sensor [ 2 ]=2 /\
IN_Handle_Sensor [ 3 ]=2))) transition from LGS_Nominal.Analog_switch_cl.Analog_switch_open to
LGS_Nominal.Analog_switch_cl.Analog_switch_closed
When LGS_Nominal if (IN_Time_to_close_switch=1 \/ (IN_Handle_Sensor [ 1 ]=0 /\ IN_Handle_Sensor [ 2
]=0 /\ IN_Handle_Sensor [ 3 ]=0)) transition from LGS_Nominal.Analog_switch_cl.Analog_switch_closed
to LGS_Nominal.Analog_switch_cl.Analog_switch_open
When LGS_Nominal if (AS_Time_to_close > AS_Time_to_close_limit) transition from
LGS_Nominal.Analog_switch_cl.Analog_switch_open to LGS_Nominal.Analog_switch_cl.failure_analog_switch
When LGS_Nominal if (AS_Time_to_open > AS_Time_to_open_limit) transition from
LGS_Nominal.Analog_switch_cl.Analog_switch_closed to
LGS_Nominal.Analog_switch_cl.failure_analog_switch
//Transition from Nominal to Analog
if (IN_anomaly[1]=1 \/ IN_anomaly[2]=1) transition from LGS_Nominal to LGS_Analog
```

Annexe D : Modèle SCT – Gridstix

```
stateMachine Gridstix

Boolean PredictFlooding
Boolean TransmitData
Boolean OrganizeNetwork
Boolean CalculateFlowRate
Boolean MeasureDepth
Boolean WiFi
Boolean Bluetooth
Boolean SPTopology
Boolean FHTopology
Boolean DistributedProcessing
Boolean SingleNodeProcessing
Float Param Batterylevel
Float Param Enum StateOfRiver [0..2]

PredictFlooding = 1
PredictFlooding * 4 = TransmitData + OrganizeNetwork + CalculateFlowRate + MeasureDepth
TransmitData = Bluetooth + WiFi
OrganizeNetwork = FHTopology + SPTopology
CalculateFlowRate = DistributedProcessing + SingleNodeProcessing
Bluetooth + DistributedProcessing <= 1

compositeState Gridstix_State
        concernLevel Energy
                simpleConstraintState No_Efficiency
                        PredictFlooding = 1
                simpleConstraintState Yes_Efficiency
                        Bluetooth=1
                        SingleNodeProcessing=1
        end_concernLevel Energy

        concernLevel FaultTolerence
                simpleConstraintState No_FaultTolerence
                        PredictFlooding = 1
                simpleConstraintState Yes_FaultTolerence
                        WiFi=1
                        FHTopology=1
        end_concernLevel FaultTolerence

        concernLevel Accuracy
```

```
                simpleConstraintState No_Accuracy
                        PredictFlooding = 1
                simpleConstraintState Yes_Accuracy
                        DistributedProcessing=1
        end_concernLevel Accuracy
end_compositeState Gridstix_State

Start transition to Gridstix_State
Start transition to Gridstix_State.Accuracy.No_Accuracy
Start transition to Gridstix_State.Energy.No_Efficiency
Start transition to Gridstix_State.FaultTolerence.No_FaultTolerence

When Gridstix_State if (Batterylevel<40) transition from Gridstix_State.Energy.No_Efficiency to
Gridstix_State.Energy.Yes_Efficiency
When Gridstix_State if (StateOfRiver=0) transition from Gridstix_State.Energy.No_Efficiency to
Gridstix_State.Energy.Yes_Efficiency
When Gridstix_State if (StateOfRiver=1 \/ StateOfRiver=2 ) transition from
Gridstix_State.Accuracy.No_Accuracy to Gridstix_State.Accuracy.Yes_Accuracy
When Gridstix_State if (StateOfRiver=2) transition from
Gridstix_State.FaultTolerence.No_FaultTolerence to Gridstix_State.FaultTolerence.Yes_FaultTolerence


When Gridstix_State if (Batterylevel>=40) transition from Gridstix_State.Energy.Yes_Efficiency to
Gridstix_State.Energy.No_Efficiency
When Gridstix_State if (StateOfRiver>0) transition from Gridstix_State.Energy.Yes_Efficiency to
Gridstix_State.Energy.No_Efficiency
When Gridstix_State if (StateOfRiver=0 ) transition from Gridstix_State.Accuracy.Yes_Accuracy to
Gridstix_State.Accuracy.No_Accuracy
When Gridstix_State if (StateOfRiver<2) transition from
Gridstix_State.FaultTolerence.Yes_FaultTolerence to Gridstix_State.FaultTolerence.No_FaultTolere
```