Un DSL pour représenter et analyser des systèmes à événements discrets *

Pascal André, Hugo Locteau, Yasmina Dali Youcef, Erwan Bousse, and Olivier Cardin

LS2N, Nantes Université, France
{firstname.lastname}@ls2n.fr

Résumé

Il existe de nombreuses approches (langages, techniques, outils) pour étudier (*i.e.* modéliser, analyser) des systèmes à événements discrets dans différents contextes. Les langages généralistes, *i.e.* indépendant du contexte d'application, ont fait leur preuve en terme d'expressivité ou d'efficacité d'analyse, mais souffrent de deux inconvénients. D'une part, ils pêchent à s'adapter à leur contexte d'application, on a donc besoin de spécialistes pour définir et analyser les systèmes. D'autre part, les outils manquent de généralité quand à l'éventail des propriétés à démontrer, ce qui amène à utiliser différents outils et nécessite donc encore plus de spécialistes pour un projet donné. Dans cet article, nous explorons l'usage de *Domain-Specific Languages* (DSLs) pour étudier des SED adaptés à leur contexte : définition d'un langage, d'un outillage de modélisation et de vérification. Cette contribution est illustrée sur une étude de cas de production manufacturière pour alimenter empiriquement la réflexion sur le sujet de l'accessibilité des langages de modélisation. Le langage est suffisamment générique pour s'adapter à d'autres contextes.

1 Introduction

Pour mettre en place des systèmes dont la dynamique est relativement complexe, on utilise des méthodes, techniques et outils fondées sur des modèles mathématiques ou informatiques. La modélisation permet d'abstraire les caractéristiques souhaitées du système pour les étudier, l'analyse du modèle permet d'en déterminer les propriétés. On se place dans une démarche d'ingénierie classique dans laquelle le processus vise à modéliser le système, vérifier ses propriétés et/ou simuler son comportement dans le temps pour estimer s'il répond aux besoins de l'organisation qui implante le système réel. Les propriétés du modèle sont sa correction ou sa validité vis-à-vis du besoin. Les propriétés fonctionnelles du système peuvent être des invariants à vérifier comme des contraintes de sûreté de fonctionnement (*safety*) ou de non blocage (*liveness*). Les propriétés non-fonctionnelles apparaissent sous forme de contraintes de qualité de service ou d'indicateurs clés de performance cible (*Key Performance Indicators, KPI*).

Nous considérons pour ce travail des Systèmes dynamiques à Evénements Discrets (SED). Les SED sont des systèmes dynamiques non linéaires à états discrets et à évolution événementielle. Contrairement aux systèmes (à états) continus décrits par des équations différentielles (temps continu) ou des équations aux différences (temps discret) contrôlés par le temps, les SED ont un espace d'états décrit par un ensemble discret [16].

Il existe de nombreuses approches (langages, techniques, outils) pour étudier (*i.e.* modéliser, analyser) des systèmes à événements discrets dans différents contextes. Les langages peuvent inclure la description explicite ou implicite d'états, de synchronisation, de concurrence ou de

*Les auteurs remercient l'Agence Nationale de la Recherche (ANR) pour son soutien financier dans le cadre du projet RODIC, numéro de subvention ANR-21-CE10-0017.

communication, inclure du temps (temporel, temporisé), ou des probabilités (stochastique). Les techniques d'analyse couvrent la vérification statique de propriétés, l'analyse symbolique, le model checking. Elle peuvent s'appuyer sur l'exploration des états possibles du systèmes ou des chemins. L'analyse peut s'appuyer aussi sur de la simulation selon des scénarios d'exécution. Les langages généralistes, *i.e.* indépendant du contexte d'application, ont fait leur preuve en terme d'expressivité (ici au sens de paradigmes pris en compte comme les états, le temps ou les événements) ou d'efficacité d'analyse.

Ces langages pour SED souffrent de deux inconvénients pour une prise en main par des non-spécialistes. D'une part, ils ne s'adaptent guère au contexte d'application, leur expressivité est limitée au modèle mathématique associé et on a besoin de spécialiste pour définir et analyser les systèmes. D'autre part, les outils ne couvrent pas l'éventail des propriétés à démontrer. Couvrir avec plusieurs outils implique encore plus de spécialistes pour un projet donné.

Dans cet article, nous explorons l'usage d'un *Domain-Specific Languages* (DSL) pour étudier des SED : définition du langage et de l'outillage de modélisation et de vérification. Le cas d'usage sous-jacent est la production manufacturière mais le langage est plus générique et peut s'adapter à d'autres contextes. Cette contribution est illustrée sur une étude de cas et permet d'alimenter empiriquement la réflexion sur le sujet en la comparant avec les approches classiques.

L'article est organisé comme suit. Nous commençons par introduire les éléments de contexte dans la section 2. La section 3 décrit la contribution principale, à savoir un DSL pour modéliser des systèmes à événements discrets, applicable notamment aux systèmes manufacturiers. La mise en œuvre de ce DSL est détaillée dans la section 4. Le langage été expérimenté sur plusieurs études de cas, nous en décrivons une dans la section 5. Nous discutons des propriétés du DSL, son positionnement dans les systèmes à événements discrets ; sa combinaison avec d'autres DSL en section 6 avant de conclure.

2 Contexte

Le contexte général de ce travail est celui de l'ingénierie système (*system engineering*) [22] au niveau de la conception abstraite. Plus particulièrement, nous nous intéressons à la modélisation de systèmes (*system modelling*) qui couvre trois points de vue complémentaires, la structure, le comportement dynamique (le contrôle) et le comportement fonctionnel (les calculs) [4, 5]. Comme indiqué dans l'introduction, on se focalise sur les systèmes à temps discret avec des événements. L'application se fait ici aux systèmes manufacturiers [11], ce qui a une certaine importance pour les caractéristiques des langages de modélisation choisis.

Formalismes et outils (l'offre)

Pour concevoir les systèmes, on a besoin d'une méthode avec des concepts (philosophie), des notations (langage), un processus (modélisation, vérification, validation, exploitation) et des outils pour automatiser les traitements sur les modèles. On parle aussi d'ingénierie des modèles pour produire les outils. La littérature sur le sujet est conséquente et de nombreuses propositions existent [4]. Leurs langages se distinguent notamment par (i) le degré de formalisme avec des méthodes formelles (réseaux de Petri RDP, systèmes de transitions, automates, modèle relationnel MR, Z, B...) ou semi-formelles (Entité-Association EAP, UML, SysML, flots de données DFD...), (ii) la couverture des aspects du système : structure (EAP, MR), dynamique (RDP, automates), fonctionnel (Z, DFD...), (iii) leur généralité (applicabilité/adéquation à des cas différents), (iv) leur propriétés ou qualités (expressivité, degré d'abstraction, modularité, correction, cohérence, complétude, testabilité, performance...), (v) l'outillage associé, etc.

Il existe des langages large spectre comme UML, SDL ou SysML, très expressifs mais dont l'outillage associé aux étapes du processus reste limité. A l'inverse les langages se focalisant sur un aspect du système dans une théorie donnée sont équipés d'outils performants de vérification mais manquent alors d'expressivité.

Domaine étudié et besoins identifiés

Le domaine d'application choisi pour cette étude est celui des systèmes manufacturiers et plus particulièrement les Reconfigurable Manufacturing Systems (RMS). Nous souhaitons modéliser des chaînes de production et analyser leur performance avant leur mise en place effective. Le point de vue est celui d'un opérateur humain : on s'abstrait donc de la structure cyber-physique de production (les machines) et du contexte organisationnel (le système d'information de l'entreprise) duquel on ne retient que les ordres de fabrication, les produits et les indicateurs de performance.

— La modélisation pour cette application doit couvrir les trois aspects du système (structure, dynamique, fonctions). L'aspect dynamique est incontournable et le contrôle, dans un système de production, suit deux logiques non-exclusives : l'approche flots de données (la chaîne est cadencée par l'arrivée des produits) et l'approche flot de contrôles (des événements modifient/perturbent le déroulement du processus de production).

— L'analyse doit couvrir à la fois la vérification formelle et la simulation, ce qui exclut les approches semi-formelles comme UML ou SysML.

— Une contrainte importante est que les acteurs du systèmes de production (responsable d'ateliers, conducteur de lignes ou opérateur) sont aussi les utilisateurs des outils de modélisation. L'expressivité est donc un critère d'acceptabilité de l'approche. Les approches purement mathématiques comme les systèmes de contrainte, les algèbres, les logiques mathématiques de base, le calcul matriciel et autres sont donc exclues.

— Une dernière contrainte du domaine est la *scalabilité*, c'est-à-dire la possibilité de passer à l'échelle de manière naturelle. On souhaite en particulier composer des systèmes de systèmes à plusieurs niveaux de profondeurs.

Alternatives et solution choisie

Parmi les approches candidates, on trouve des systèmes à événements discrets (SED) de deux types : les outils de vérification formelle et les simulateurs à événements discrets.

Bérard et al. [7] proposent un éventail de langages et outils pour la modélisation et la vérification par *model-checking* de la dynamique des systèmes dits réactifs tels que les réseaux de Petri, les automates communicants ou les algèbres de processus. Ils peuvent, ou pas, prendre en compte le temps réel et des paramètres plus riches tels que des objets. Notons que la vérification formelle par *model-checking* peut se faire à partir de l'exploration de l'espace d'états du système et/ou l'analyse des chemins (traces) par des logiques adaptées comme les logiques temporelles. Les traces peuvent être obtenues par simulation (exécutions indépendantes les unes des autres) ou par parcours de l'espace d'états du système (les chemins produisent les traces). Nous retenons principalement les automates et les réseaux de Petri comme formalisme visuel accessible aux opérateurs, nous avons testé Romeo [14], Tina [8] et CPN Tools [21]. L'inconvénient majeur des réseaux de Petri est le manque de modularité pour réutiliser des parties de modèles dans d'autres modèles, ce qu'un opérateur souhaite faire lors de la conception de la chaîne de production. Les automates synchronisés ou communicants (ou variantes de systèmes de transition) sont plus modulaires et a priori adaptés mais les outils existants tels que UPPAAL [6] restent limités

dans l'expressivité des données et des calculs. Les algèbres de processus sont plus expressives, notamment leur extensions comme LOTOS et l'outil CADP [13], mais la description reste technique et le découpage pas toujours naturel. Un manque récurrent est la possibilité de mixer flots de données et flots de contrôle dans les outils formels.

Jadri et al. comparent des outils de simulation d'événements discrets dans un environnement académique [17]. Dans nos travaux, nous avons utilisé Arena [18] et FlexSim [20]. Une comparaison entre Simio et Arena est proposée dans [23]. Ces outils sont puissants à la fois pour l'expressivité avec des IHM très réalistes et pour la simulation avec du calcul d'indicateurs. Par contre le niveau d'expertise est souvent élevé (description structurelle vs flots de contrôles) et le niveau d'abstraction bas (nombreux paramètres très cachés dans les sous-fenêtres d'IHM).

Nous n'avons pas trouvé un trouvé d'outil couvrant l'ensemble des besoins entre expressivité, simplicité, adaptation au domaine manufacturier, avec vérification et simulation. Nous avons donc décidé de développer notre propre langage, plus léger, adapté au domaine des systèmes manufacturiers reconfigurables. On qualifie ces langages de *domain specific* ou DSL.

3 Un DSL pour modéliser les systèmes manufacturiers

Cette section présente conceptuellement un DSL pour modéliser et étudier des RMS pour lesquels des descriptions de type SED modulaires sont requises. Nous résumons d'abord la syntaxe de ce DSL, dans sa forme dite abstraite définie par un métamodèle et dans sa forme dite concrète définie par une grammaire. Nous résumons ensuite la sémantique opérationnelle de ce DSL, à savoir la définition de l'état de simulation et de son évolution dans le temps. Notez que, pour des raisons de place, nous n'exposons ici qu'un extrait du métamodèle et une version détaillée figure en Section A.1 de l'annexe que nous avons mise à disposition en ligne [1].

3.1 Syntaxe abstraite

La syntaxe abstraite du DSL proposé est définie sous la forme d'un métamodèle, à savoir un modèle orienté objet définissant un ensemble de concepts sous forme de classes. Ce métamodèle est structuré en deux parties : d'abord les concepts permettant la définition de la *structure* d'un système, à savoir l'assemblage des modules du système de production modélisé, puis les concepts permettant la définition du *comportement* dynamique et fonctionnel des modules, à savoir l'évolution dans le temps et les traitements réalisés.

Pour définir la structure d'un système, le concept principal est nommé ProductionSystem, qui décrit une configuration du système de production. La modularité est une caractéristique majeure des RMS [2]. Pour la mettre en œuvre, le modèle est organisé en Modules définis par des ModuleTypes. Un extrait [3] du métamodèle structurel illustre ces concepts dans la FIGURE 1. Un même module type peut donc être instancié plusieurs fois dans le système, avec des paramètres différents. Les modules sont reliés entre eux par des liens Binding via des connecteurs (Connect**ors**), qui peuvent être soit "offerts" (c'est à dire, mentionnent une caractéristique sortante) ou "requis" (c'est à dire, mentionnent une caractéristique entrante). Des Elements (produits, conteneurs, événements) transitent par ces liens. Selon l'interprétation choisie, on peut souhaiter travailler en flot de données (produits) ou flot de contrôle. Un module dispose de différentes Zones pour stocker ces éléments. Il peut rendre accessible ses zones à d'autres

1. https://uncloud.univ-nantes.fr/index.php/s/2depFJSeJ6RkeiR
2. Les *Reconfigurable Manufacturing Systems* sont une classe de systèmes manufacturiers dans laquelle la production peut évoluer selon le contexte (demande, aléas techniques, pannes...) [19].
3. La structure est détaillée dans FIGURE 8 de l'annexe web[1].

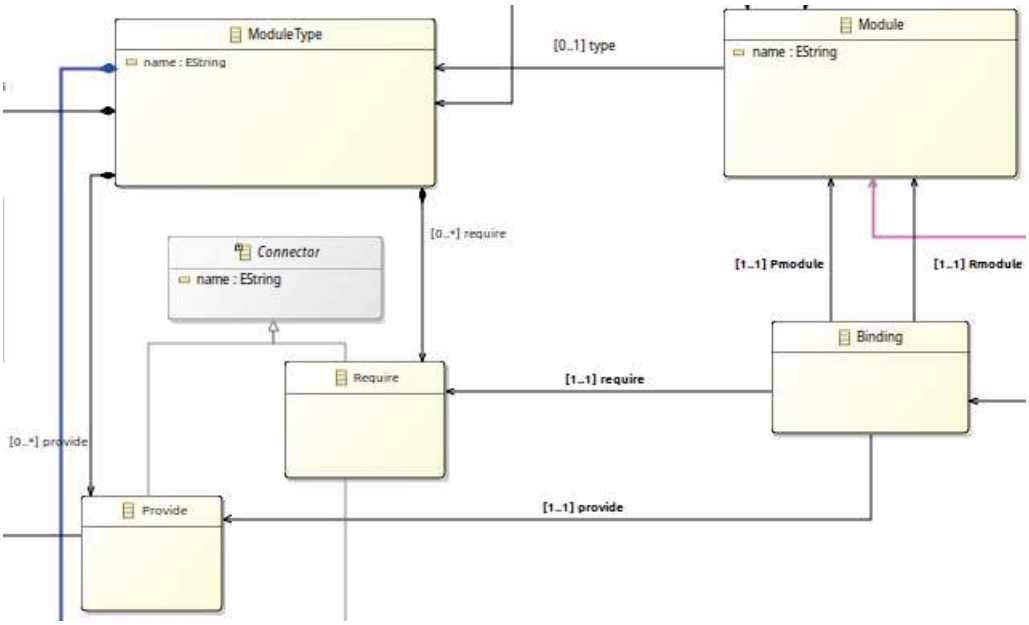

FIGURE 1 – Extrait de la syntaxe abstraite (`Module`, `ModuleType` et `Connector`)

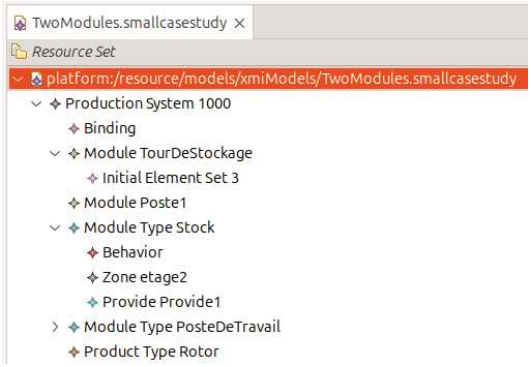

FIGURE 2 – Exemple partiel de modèle conforme à la syntaxe abstraite du DSL, sous forme arborescente

```
modules {
    Module TourDeStockage {
        type Stock
        initialelementsets {
            InitialElementSet {
                stock 3
                elementtype "Rotor"
                zone "Stock.etage2"
            }
        }
    } ,
    Module Poste1 {
        type PosteDeTravail
    }
}
```

FIGURE 3 – Représentation textuelle partielle du modèle montré en FIGURE 2

modules via les connecteurs pour l'échange d'éléments. La structure modulaire de la configuration facilite l'évolution du système. La FIGURE 2 montre partiellement sous forme arborescente un exemple de modèle conforme à la partie structurelle de la syntaxe abstraite du DSL. La racine du modèle est une instance du concept ProductionSystem, auquel est associé une durée de simulation souhaitée de 1000 unités de temps. Deux ModuleTypes sont définis, un premier appelé `Stock` qui décrit la structure et le comportement d'une zone de stockage, et un second appelé `PosteDeTravail` qui décrit la structure et le comportement d'un poste de travail d'assemblage. Deux Modules sont définis, un premier appelé `TourDeStockage` instance du type `Stock`, et un second appelé `Poste1` instance du type `PosteDeTravail`. Un Binding connecte ces deux modules.

Le comportement dynamique et fonctionnel (Behavior) des modules est défini dans leur type (ModuleType) sous forme de tâches dont le déclenchement est défini par des règles (TaskRule). Cette partie comportementale du modèle est donnée dans FIGURE 9 de l'annexe web[1]. Il n'y a donc pas de notion explicite d'état (et état courant) comme dans un automate ou un réseau de Petri. Une TaskRule spécifie une unité de logique d'exécution, incluant les conditions de déclenchement de la règle Precondition et les Actions à effectuer. Une précondition définit les conditions logiques qui doivent être remplies pour que la règle de tâche correspondante soit éligible à l'exécution. En particulier, on peut tester la présence d'un Element ayant un état donné dans une Zone donnée. Chaque TaskRule encapsule une ou plusieurs actions qui définissent la sémantique opérationnelle du comportement. Le DSL prend en charge diverses actions spécifiques au domaine permettant la modélisation de comportements tels que la création, le déplacement et la reconfiguration structurelle des Elements.

3.2 Syntaxe concrète textuelle

La syntaxe concrète textuelle du DSL est directement destinée aux utilisateurs du DSL, et permet de définir des modèles conformes sous forme de texte. Elle est définie sous la forme d'une grammaire qui structure textuellement un modèle principalement à l'aide d'accolades et de mots-clés associés aux différents concepts de la syntaxe abstraite. Cette grammaire permet de produire un analyseur syntaxique capable de traduire un modèle textuel en un arbre syntaxique conforme à la syntaxe abstraite présentée précédemment.

La FIGURE 3 montre partiellement sous forme textuelle le modèle présenté initialement sous forme arborescente en FIGURE 2. Chaque module est déclaré à l'aide du mot-clé `Module`, et le type d'un module est déclaré à l'aide du mot-clé `type`. Le mot-clé `InitialElementSet` permet de définir un stock initial de produits présents au sein d'une zone d'un module.

3.3 Sémantique opérationnelle

La sémantique opérationnelle du DSL définit l'état de simulation du système et son évolution dans le temps. Une simulation est une suite d'états, qui pourra *in fine* être stockée dans une trace d'exécution de la simulation. Les changements d'états se font par les tâches inscrites dans un échéancier et qui ont une certaine durée.

L'état de simulation est défini sous la forme d'un métamodèle distinct de la syntaxe abstraite, visible au sein de la FIGURE 10 de l'annexe web[1]. Le concept principal est appelé ProductionSystemSimulation, qui maintient le contexte de simulation, y compris le système actif actuel et le temps de simulation. Il contient des instances les états d'entités telles que Product, Container, ModuleState et ZoneState, qui reflètent la configuration réelle, avec l'emplacement des produits et les conditions du module. Le comportement d'exécution est représenté par les instances Task, qui suivent des attributs tels que startTime, les comportements associés et l'état d'exécution (par exemple, `OFF`, `ON`). L'inclusion du ZoneState permet un suivi détaillé du placement des éléments dans les modules.

La sémantique opérationnelle du DSL définit également de quelle manière un tel état de simulation évolue dans le temps. On détermine la suite des tâches qui déclenchent les changements d'état (plusieurs tâches peuvent se terminer au même moment). Lors d'un changement d'état, on détermine les TaskRules applicables et donc les tâches à entrer dans l'échéancier de simulation. Lorsque les tâches se terminent dans la chronologie du temps simulée, les éléments et produits concernés par ces tâches sont déplacés de zones en zones (dont les contenus sont capturés par des objets ZoneState) comme spécifiés dans les comportements des ModuleTypes.

```java
public class ModuleImpl extends MinimalEObjectImpl.Container implements smallcasestudy.Module {
    /**
     * The cached value of the '{@link #getType() <em>Type</em>}' reference.
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @see #getType()
     * @generated
     * @ordered
     */
    protected ModuleType type;

    /**
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @generated
     */
    public ModuleType getType() {
        if (type != null && type.eIsProxy()) {
            InternalEObject oldType = (InternalEObject) type;
            type = (ModuleType) eResolveProxy(oldType);
            if (type != oldType) {
                if (eNotificationRequired())
                    eNotify(new ENotificationImpl(this, Notification.RESOLVE, SmallcasestudyPackage.MODULE__TYPE,
                            oldType, type));
            }
        }
        return type;
    }
```

FIGURE 4 – Extrait de l'API Java générée à partir du métamodèle Ecore montré en FIGURE 1, avec la classe Java pour le concept `Module`

Une exécution complète de modèle à l'aide de la sémantique opérationnelle peut être mémorisée sous la forme d'une trace d'exécution, à savoir la suite des états de simulation, estampillés par le temps de simulation. Une telle trace peut ensuite être analysée pour étudier les propriétés du système modélisé.

4 Implantation du DSL proposé

Dans cette section, nous décrivons comment nous avons implémenté le DSL décrit conceptuellement en Section 3 afin d'obtenir un outillage complet utilisable pour un expert du domaine. Nous avons utilisé ECLIPSE GEMOC STUDIO [4] [9, 12], un atelier de création de DSLs basé sur l'écosystème *Eclipse Modeling Framework* (EMF) [5], qui présente plusieurs avantages, notamment celui d'intégrer plusieurs outils compatibles tels que le langage de métamodélisation Ecore, le framework Xtext permettant de définir une syntaxe concrète textuelle, le framework Sirius permettant de définir une syntaxe concrète graphique, ou encore le langage de métaprogrammation K3 dédié à la définition d'une sémantique opérationnelle. En outre, ECLIPSE GEMOC STUDIO fournit un ensemble d'outils de support à l'exécution de modèles, en particulier un débogueur de modèles et un constructeur de traces d'exécutions.

4.1 Syntaxe *abstraite* et d'un éditeur arborescent de modèles

La syntaxe abstraite du DSL a été implémentée à l'aide du langage de métamodélisation Ecore. Ce langage permet de définir graphiquement un ensemble de classes, des attributs de classes, et des liens entre celles-ci, comme les références, la composition ou encore l'héritage. La FIGURE 1 présentée précédemment montre un extrait du métamodèle du DSL définit à l'aide du langage graphique Ecore. Une force du langage Ecore est être accompagné de l'environnement

4. https://gemoc.org/studio.html
5. https://eclipse.dev/emf/

EMF, qui permet notamment de générer automatiquement une API Java complète permettant de créer, modifier, sauvegarder et charger un modèle conforme au métamodèle considéré. Cette API générée est principalement composée d'une ensemble de classes Java, chacune étant générée à partir d'une classe du métamodèle Ecore. La FIGURE 4 montre un extrait de code généré pour la classe Module. En outre, il est possible de générer à partir du métamodèle un éditeur arborescent permettant d'éditer un modèle sous forme d'arbre. Un modèle peut alors être sérialisé sous forme d'un fichier à l'aide du format standard XMI (basé sur le format XML). La FIGURE 2 présentée précédemment montre un extrait de modèle en cours d'édition dans l'éditeur arborescent généré pour le DSL.

4.2 Syntaxe *concrète* textuelle et d'un éditeur associé

L'éditeur arborescent généré présente des inconvénients, comme celui d'être difficile à utiliser pour de larges modèles, rendant la compréhension du modèle par l'utilisateur difficile. Cette limitation est l'une des raisons pour lesquelles le DSL a été adjoint d'une syntaxe concrète textuelle, dans le but de pouvoir éditeur les modèles directement sous forme de texte (ou, dit autrement, de code).

La syntaxe concrète textuelle du DSL a été implémentée à l'aide du framework Xtext. Cette implémentation prend principalement la forme d'une grammaire écrite à l'aide du langage Xtext, à laquelle est associé un ensemble de modules écrits en Java pour préciser certaines caractéristiques de la syntaxe (la portée des identifiants, etc.) et du fonctionnement de l'éditeur textuel souhaité (autocomplétion, règles de validation de modèles, etc.). Le grand avantage d'Xtext est sa capacité, à partir d'une grammaire, de générer automatiquement non seulement un analyseur syntaxique capable de charger en mémoire un modèle textuel, mais également un éditeur textuel complet de modèles intégré à Eclipse. Un éditeur généré par Xtext vient directement avec une coloration syntaxique, la possibilité de naviguer entre éléments, un plan du code en cours d'édition, et un outil d'autocomplétion.

La FIGURE 3, présentée précédemment, montre un extrait de modèle représenté textuellement à l'aide de la syntaxe concrète, ouvert au sein de l'éditeur de modèle généré à l'aide d'Xtext. On peut voir les mots-clés mis en évidence par la coloration syntaxtique.

4.3 Sémantique opérationnelle en simulateur de modèles

La sémantique opérationnelle du DSL a été implémentée à l'aide du langage de métaprogrammation K3. Ce programme K3 peut exécuter un modèle conforme à la syntaxe du DSL à l'aide de l'API Java générée précédemment à partir du métamodèle Ecore (voir Section 4.1). Une sémantique K3 se compose d'une opération pour initialiser l'état d'exécution initial d'un modèle (à l'aide d'une annotation `@InitializeModel`), d'une opération principale pour démarrer l'exécution d'un modèle (à l'aide d'une annotation `@Main`), et d'un ensemble d'opérations qui définissent les pas d'exécutions qui font avancer la simulation (à l'aide d'une annotation `@Step`). Ces annotations sont automatiquement reconnues par ECLIPSE GEMOC STUDIO lors de l'exécution d'un modèle, et permettent à l'environnement d'être informé lorsque chaque pas d'exécution se produit. Ce mécanisme (décrit plus en détail dans [10]) permet entre autre à ECLIPSE GEMOC STUDIO de générer automatiquement une trace d'exécution qui capture la séquence d'états atteints par un modèle durant son exécution.

La sémantique opérationnelle du DSL est *in fine* un programme Java qu'on peut ensuite exécuter, dans un framework lié à Eclipse GEMOC Studio.

5 Expérimentation de l'outil

Dans cette section, nous décrivons l'étude cas support, sa mise en œuvre dans notre DSL et les premiers résultats de simulation.

5.1 Etude de cas

Le système manufacturier considéré ici est imaginaire. Il a été conçu et étudié via le logiciel FlexSim comme détaillé dans la section A.5 de l'annexe web[1]. Il est principalement constitué d'une boucle de convoyeurs à accumulation sur laquelle circulent des palettes. Les palettes sont toujours sur le convoyeur, et avancent à vitesse constante de 1m/s. Des bloqueurs sont installés devant les emplacements de travail afin de permettre aux opérations de production de se dérouler dans de bonnes conditions mécaniques. Le premier poste de travail est équipé d'un robot polyarticulé 6 axes. Ce robot prend un par un des produits dans un stock placé à son pied lorsqu'une palette est disponible et arrêtée au poste. Prendre un produit nécessite 5 secondes, le déposer sur la palette 5 secondes supplémentaires. Les produits arrivent dans le stock un par un, séparés par un temps inter-arrivée suivant une loi normale, de moyenne 14.5s et d'écart-type 3s. Le robot place 2 produits sur chaque palette, puis la palette est libérée et avance sur le convoyeur Au second poste de travail, situé 15m50 après le premier, un opérateur humain prend un produit, puis va le déposer sur une machine de traitement 1 située à 5m du point de prise. La palette est libérée dès que l'opérateur a pris le produit et avance au poste suivant. L'opérateur traite le produit sur la machine, puis vient la déposer sur la palette, qui a donc avancé entre temps. Au troisième poste de travail, situé 6m après le second, un opérateur humain prend le second produit, puis va le déposer sur une machine de traitement 2 située à 5.6m du point de prise. L'opérateur traite le produit sur la machine, puis revient la déposer sur la palette. Lorsque la palette a reçu les 2 produits, elle est libérée et avance au poste suivant. Au quatrième poste de travail, situé 5m10 après le troisième poste, un chariot de transfert vient retirer les 2 produits un par un, et les dépose dans un stock final. L'opération totale dure 18s (arrivée de la palette pleine jusqu'au départ de la palette vide). Lorsque la palette est vide, elle est libérée et retourne au premier poste, situé à 10m du quatrième poste de travail.

5.2 Modélisation

Le modèle est défini comme un assemblage de modules M_i, chacun disposant de zones Z_i qu'il peut partager via ses connecteurs. La FIGURE 5 met en évidence ces modules et les zones associées ; les modules décrivant les postes de travail mais aussi les éléments modulaires du convoyeur à bloqueurs. Notre métamodèle RMS a permis de modéliser le système décrit ci-dessus de la façon suivante : (1) Les 4 palettes sont présentes à l'état initial dans une zone d'un module nommé Source1. (2) Un module Source 2 génère toutes les 14 secondes un nouveau produit. (3) Le convoyeur est divisé en plusieurs modules sur lesquels les palettes transitent. (4) Les postes de travail correspondent à un module composé de 2 zones. Le produit change d'état en passant d'une zone à l'autre. (5) A la fin de leur parcours, les produits atteignent la zone Sink d'un module dédié. Le Listing 1 de l'annexe web[1] donne la représentation textuelle.

5.3 Evaluation, traces...

Des logs ont été ajoutés à la sémantique pour nous permettre de vérifier que le comportement du système à l'exécution est bien celui attendu, et ainsi de valider notre modélisation. Ils

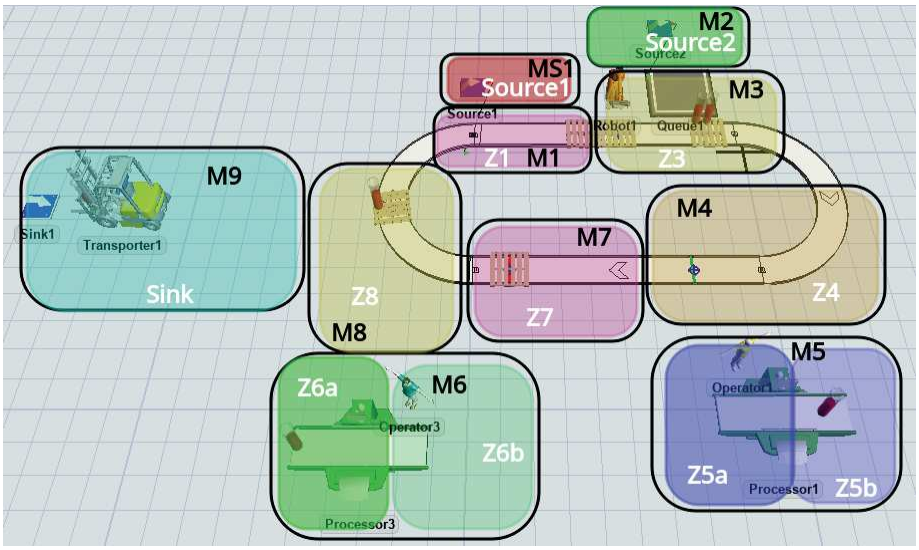

FIGURE 5 – Modélisation du système de production

seront désactivés dans un environnement de production. Pour chaque pas d'exécution les informations transmises par les logs sont les suivantes : (1) temps du système (exprimé en secondes), (2) déplacement d'un élément (produits ou conteneurs) d'une zone à l'autre, (3) déplacement d'un produit dans ou hors d'un conteneur, (4) modification de l'état d'un produit.

A titre d'exemple, pour les conditions suivantes ((1) durée de simulation = 2500 secondes, (2) génération d'un produit toutes les 14 secondes dans la source2, (3) déplacement d'un produit de 20 seconds de la zone Z5b à la zone Z7 et (4) déplacement d'un produit de 10 secondes entre les autres zones), nous obtenons les résultats suivants : (a) 76 produits ont atteint la zone Sink—M9. (b) 96 produits sont en attente dans le stock Source2—M2. (c) Le module M5 fonctionne durant 1982 secondes et et M6 1558 secondes. (d) Le temps de traitement des produits ayant l'identifiant 20 est de 140 secondes. C'est le temps mis par ce produit pour quitter la zone Source2 et atteindre la zone Sink. Ces informations peuvent également être déduites de la lecture du fichier *trace* généré pendant l'exécution grâce au plugin `simple.trace`. S'agissant d'un fichier XMI, il peut être lu par un humain dans l'éditeur d'arbre du ECLIPSE GEMOC STUDIO, mais cette tâche est assez complexe, en particulier pour les grandes simulations. Pour une vérification pas à pas du mouvement d'un élément, les journaux sont plus pratiques. Cependant, lorsque le produit logiciel final sera validé et publié, les journaux d'exécution seront désactivés de la sémantique. Le fichier *trace* sera interrogé à l'aide d'un autre DSL appelé TraceDQL [1], ou d'un DSL d'extraction de KPI spécifique qui reste à concevoir, afin de fournir à l'opérateurs l'évaluation de performance de sa simulation.

6 Discussion

Dans cette section, nous discutons des apports et limites de la proposition, vis-à-vis d'autres formalismes/outils utilisés dans les SEDs selon le positionnement fixé dans la section 2. Ces outils sont les outils de vérification formelle et les simulateurs à événements discrets, et ont été introduit dans la Section 2.

6.1 Comparaison avec les outils de vérification formelle

Pour permettre la vérification du modèle, nous utilisons habituellement des outils puissants pour la dynamique des systèmes tels que Tina [8], Romeo [14], UPPAAL [6] ou SPIN [15]. Nous illustrons ici le cas des Réseaux de Petri (RdP) avec Tina, d'autre exemples sont donnés dans la Section A.3 de l'annexe web[1].

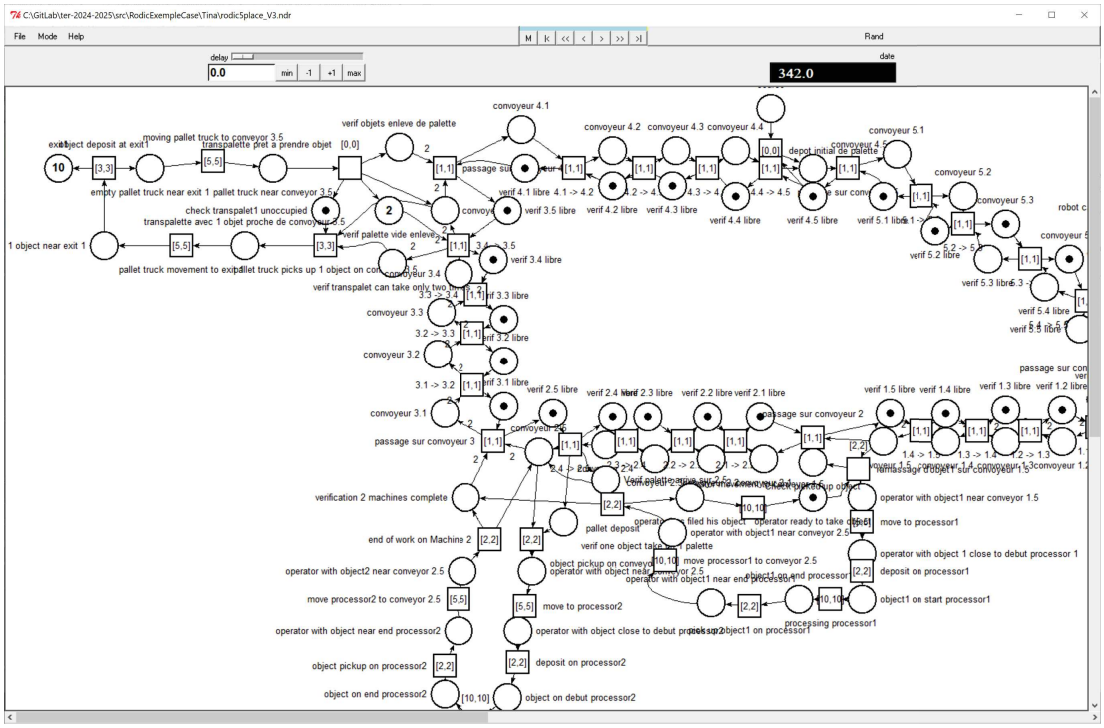

FIGURE 6 – Cas d'étude simplifié traité avec Tina

La FIGURE 6 illustre une simulation du cas avec l'outil Tina (le modèle est donné dans la FIGURE 11 de la section A.3 l'annexe web[1].). On y retrouve distinctement les 4 postes ainsi que les 5 convoyeurs formant un système circulaire. Chaque convoyeur est représenté par 5 places pour 5 emplacements de palettes, chaque place d'un convoyeur dispose d'une place de garde supplémentaire permettant de limiter le passage des palettes. Cette garde empêche les palettes de se superposer dans le système en créant une file de palettes. Le temps est ajouté à chaque transition indiquant la durée de chaque étape. Tina n'utilisant que le formalisme des jetons simples le flux des palettes sans objets se déroule avec un seul jeton et une fois que deux objets sont sur les palettes ce sont deux jetons qui se déplacent simultanément de place en place. Une fois le modèle réalisé dans Tina, il est possible de vérifier les propriétés du RDP grâce au outils de vérification de Tina. Nous ne détaillons pas ici les résultats, le lecteur pourra consulter la section A.3 de l'annexe web[1].

Concernant la simulation, l'intérêt de Tina par rapport aux autres outils testés est qu'il conserve une horloge globale de simulation pour le temps. Il permet de plus de simuler pas à pas ou globalement. En tenant compte d'une règle aléatoire pour les intervalles de temps, on obtient un temps de simulation, ici on obtient 342 avec 3 palettes et 10 produits.

6.2 Comparaison avec un langage de spécification large spectre

Kmelia [2] est un langage permettant de décrire un modèle de composant basé sur la description de services complexes. Les composants Kmelia sont abstraits, indépendants de leur environnement et donc non exécutables. Kmelia peut être utilisé pour modéliser des architectures logicielles et leurs propriétés, ces modèles étant ensuite affinés pour devenir des plates-formes d'exécution. Il peut également servir de modèle commun pour étudier les propriétés des modèles de composants ou de services (abstraction, interopérabilité, composabilité). COSTO est l'outil associé à Kmelia, il a été développé en Java.

Pour l'étude de cas, chaque poste est représenté par un composont Kmelia, dont les services sont les opérations sur les produits. Le convoyeur est un composant muni de services "bloqueurs" pour échanger les palettes et les produits avec les compostants "Postes". Le convoyeur est spécifié par un tableau qui implante la file circulaire. L'ensemble des services étant concurrent, ils sont implantés par des *threads* et l'état des composants implique des synchronisations pour conserver la cohérence des objets. La multiplication des *threads* pose des problèmes de blocages lorsque trois postes sont spécifiés. Ces problèmes sont dus à la version actuelle de la sémantique opérationnelle Java dans COSTO, rendant cette approche inexploitable dans l'immédiat. Bien que Kmelia soit plus expressif que la plupart des outils d'analyse formelle mentionnés dans cette section, il reste limité : on a besoin de types de données plus sophistiqués que les ensembles et tableaux pour représenter le comportement fonctionnel. Des détails de la modélisation et de l'évaluation dans COSTO/Kmelia sont donnés dans la section A.4 de l'annexe web[1].

6.3 Comparaison avec des simulateurs

Une simulation de ce système a été réalisée avec le logiciel FlexSim [6]. Flexsim est un environnement logiciel orienté objet utilisé pour développer, modéliser, simuler, visualiser et contrôler les activités et les systèmes de flux dynamiques [20]. L'environnement Flexsim est complètement intégré au compilateur C++ et utilise flexscript (une bibliothèque C++ précompilée) ou C++ directement. Toutes les animations sont OpenGL et offrent un rendu attractif en 2D ou en 3D. Flexsim a été utilisé pour modéliser des processus de fabrication ou de logistique.

FlexSim fournit une interface graphique conviviale pour décrire les systèmes de fabrication et une simulation visuelle qui sont également très pratiques pour être observés par des utilisateurs non experts parce que des éléments physiques ainsi que la mise en page sont impliqués. En contrepartie, l'effort de modélisation est important et nécessite une solide expérience, en particulier pour la partie relative au contrôle des flux. En outre, le modèle n'est pas modulaire, malgré les possibilités de regroupement, car le flux de contrôle est unique. L'approche DSL permet plusieurs niveaux d'abstraction et d'évolutivité par la composition et l'encapsulation de modules. Les détails de la modélisation et de l'évaluation de FlexSim sont donnés dans la section A.5 de l'annexe web[1].

Nous avons observé des valeurs KPI similaires entre la simulation des paramètres déterministes de FlexSim et les calculs DSL/trace. Bien que l'environnement FlexSim soit beaucoup plus puissant, nous constatons que son usage reste l'apanage de spécialistes et l'investissement beaucoup plus important par rapport à notre DSL.

6.4 Complémentarité

Nous ne nous positionnons pas comme des concurrents des outils mentionnées dans cette section, qui sont bien plus puissants et performants. Avec un DSL, le positionnement vise à in-

6. https://www.flexsim.com/

troduire l'utilisateur final au centre de l'activité, à fournir des fonctions de base suffisantes pour étudier rapidement la performance (KPI) de leurs systèmes. Les concepts sont personnalisés et donc plus facilement appréhendables par l'utilisateur final. Plus exactement, nous suggérons plusieurs langages : un pour les modèles, un pour les performances, un pour les scénarios de simulation, un pour les propriétés, etc. Pour vérifier des propriétés, l'idée, que nous avions déjà développée avec Kmelia [2] pour les propriétés dynamiques et les contrats : les DSL servent de pivot pour traduire vers des outils dédiés selon les propriétés à vérifier. Chaque outils a en effet des capacités à traiter telle ou telle propriété. La transformation vers les réseaux de Petri nous semble assez naturelle (les *zones* sont des places, les *tasks* sont des transitions) et permettra de couvrir la vérification formelle de propriétés non couvertes par la simulation. Inversement, une traduction vers FlexSim semble très compliquée car c'est un langage très opérationnel s'appuyant sur des bibliothèques de composants de base spécifiques.

7 Conclusion

De nombreux outils existent pour représenter et analyser des systèmes à événements discrets. D'une part, les outils généralistes à base d'automates, de réseaux de Petri ou d'algèbres de processus s'appuient sur des bases formelles pour vérifier les propriétés, ils s'appuient aussi sur la simulation pour les propriétés temporelles. Dans les deux cas, une certaine expérience est nécessaire pour maîtriser la preuve. D'autre part, les outils de simulation à base d'agents, sont plus expressifs mais nécessitent aussi de l'expertise pour décrire la logique et calculer les indicateurs de performance. Nous proposons dans cet article une approche plus légère, dont l'intérêt majeur est de s'adapter au domaine ciblé, notamment pour nommer les concepts du langage. On cible des utilisateurs plus spécialistes du domaines que des techniques de modélisation et de preuve. Nous avons montré comment écrire un tel langage et l'exploiter sur un exemple de système manufacturier de type *flow-shop* pour en évaluer la performance.

Les perspectives de ce travail sont nombreuses. A court terme, nous devons développer l'expressivité du langage en termes de calculs, mais aussi enrichir la spécification des éléments de simulation pour une exploitation personnalisée via le langage de traces. En particulier, un langage pour décrire les scénarios de simulation est à l'étude. Concernant la vérification formelle, notre vision est celle de la réutilisation des outils performants via des transformations de modèles pour générer les spécifications, par exemple dans les RDP ou les automates, mais surtout d'un langage pour exprimer les propriétés, qui devront elles aussi être transformées pour être exploitables dans les outils mentionnés.

Références

[1] Hiba Ajabri, Jean-Marie Mottu, Christian Attiogbé, and Pascal Berruet. Navigating the trace of executable domain specific languages through a trace domain query language. In Proceedings of the European Conference on Modelling Foundations and Applications (ECMFA 2025), June 2025.

[2] Pascal Andre, Gilles Ardourel, and Christian Attiogbé. Kmelia : un modèle abstrait et formel pour la description et la composition de compo-sants et de services. Revue des Sciences et Technologies de l'Information, 30(6) :627–658, 2011.

[3] Pascal André and Olivier Cardin. A core reference model for applicable reconfigurable manufacturing systems. In Theodor Borangiu et al., editor, Proceedings of SOHOMA 2023, volume 1136 of Studies in Computational Intelligence, pages 507–519. Springer, 2023.

[4] Pascal André and Alain Vailly. Conception de systèmes d'information ; Panorama des méthodes et des techniques, volume 1 of Collection Technosup. Editions Ellipses, 2001. ISBN 2-7298-0479-X.

[5] Pascal André and Alain Vailly. Spécification des logiciels ; Deux exemples de pratiques récentes : Z et UML, volume 2 of Collection Technosup. Editions Ellipses, 2001. ISBN 2-7298-0774-8.

[6] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL—a tool suite for automatic verification of real-time systems. Springer, 1996.

[7] Béatrice Bérard, Michel Bidoit, François Laroussinie, Antoine Petit, and Philippe Schnoebelen. Vérification de logiciels : techniques et outils du model-checking. Vuibert Informatique. Vuibert, 1999. ISBN 2-7117-8646-3.

[8] Bernard Berthomieu, P-O Ribet, and François Vernadat. The tool TINA -construction of abstract state spaces for Petri nets and time Petri nets. International journal of production research, 42(14) :2741–2756, 2004.

[9] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution framework of the gemoc studio (tool demo). In Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, SLE '16, page 84–89. ACM, October 2016.

[10] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution framework of the gemoc studio (tool demo). In Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, pages 84–89, 2016.

[11] G. Chryssolouris. Manufacturing Systems : Theory and Practice. Mechanical Engineering Series. Springer, 2006.

[12] Benoit Combemale, Olivier Barais, and Andreas Wortmann. Language engineering with the gemoc studio. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pages 189–191. IEEE, 2017.

[13] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2010 : A toolbox for the construction and analysis of distributed processes. In International conference on tools and algorithms for the construction and analysis of systems, pages 372–387. Springer, 2011.

[14] Guillaume Gardey, Didier Lime, Morgan Magnin, and Olivier Roux. Romeo : A tool for analyzing time Petri nets. In International Conference on Computer Aided Verification, pages 418–423. Springer, 2005.

[15] Gerard J Holzmann. The SPIN model checker : Primer and reference manual, volume 1003. Addison-Wesley Reading, 2004.

[16] R. Husson, C. Iung, J.F. Aubry, J. Daafouz, and D. Wolf. Automatique : Du cahier des charges à la réalisation de systèmes. EEA. Dunod, 2017.

[17] Mario Jadrić, Maja Ćukušić, and Antonia Bralić. Comparison of discrete event simulation tools in an academic environment. Croatian Operational Research Review, 5(2) :203–219, December 2014.

[18] Kassu Jilcha, Esheitie Berhan, and Hannan Sherif. Workers and machine performance modeling in manufacturing system using arena simulation. Journal of Computer Science & Systems Biology, 8(4) :185–190, 2015.

[19] Yoram Koren, Xi Gu, and Weihong Guo. Reconfigurable manufacturing systems : Principles, design, and future trends. Frontiers of Mechanical Engineering, 13(2) :121–136, November 2017.

[20] W.B. Nordgren. Flexsim simulation environment. In Proceedings of the Winter Simulation Conference, volume 1, pages 250–252 vol.1, 2002.

[21] Anne Vinter et al. Ratzer. CPN tools for editing, simulating, and analysing coloured Petri nets. In International conference on application and theory of petri nets, pages 450–462. Springer, 2003.

[22] A.P. Sage. Systems Engineering. Wiley Series in Systems Engineering and Management. Wiley, 1992.

[23] António Vieira, Luís Dias, Guilherme Pereira, and José A Oliveira. Comparison of simio and arena simulation tools, 2014.