JSON application programming interface for discrete event simulation data exchange

JSON Application Programming Interface for Discrete Event Simulation data exchange

Ioannis Papagiannopoulos

Enterprise Research Centre

Faculty of Science and Engineering

Design and Manufacturing Technology

University of Limerick

Submitted to the University of Limerick for the degree of

Master of Engineering 2015

1. Supervisor: Prof. Cathal Heavey
Enterprise Research Centre
University *of* Limerick
Ireland

Abstract

This research is conducted as part of a project that has the overall aim to develop an open source discrete event simulation (DES) platform that is expandable, and modular aiming to support the use of DES at multi-levels of manufacturing companies.

The current work focuses on DES data exchange within this platform. The goal of this thesis is to develop a DES exchange interface between three different modules: (i) ManPy an open source discrete event simulation engine developed in Python on the SimPy library; (ii) A Knowledge Extraction (KE) tool used to populate the ManPy simulation engine from shop-floor data stored within an Enterprise Requirements Planning (ERP) or a Manufacturing Execution System (MES) to allow the potential for real-time simulation. The development of the tool is based on R scripting language, and different Python libraries; (iii) A Graphical User Interface (GUI) developed in JavaScript used to provide an interface in a similar manner to Commercial off-the-shelf (COTS) DES tools.

In the literature review the main standards that could be used are reviewed. Based on this review and the requirements above, the data exchange format standard JavaScript Object Notation (JSON) was selected. The proposed solution accomplishes interoperability between different modules using an open source, expandable, and easy to adopt and maintain, in an all inclusive JSON file. In this way, the DES API model offers the means for the configuration, synchronization, and maintenance of the different DES platform instances in a decentralized manner.

The presented exchange data model offers the means to represent (i) any results returned by the simulation engine, (ii) configuration data for various platform instances, and (iii) extra input data collected from various data sources. In addition, the DES API data model offers an embedded JSON schema for the validation of the DES model graphs, and for the configuration of any GUI used for DES model design. A full description of the JSON DES API data model is provided with the different elements of the interface detailed. The DES API data model consists of six parts: (i) an element which is used for the configuration and the synchronization of the platform; (ii) an embedded JSON schema for the creation of consistent models; (iii) a graph that contains the topology of the model (iv) an element that contains DES related data, such as Bills Of Materials (BOM), or Work In Progress (WIP); (iv) an element that contains the results returned by the simulation engine; (v) and an element that contains general information crucial for a simulation run (e.g. maximum simulation run time, simulation time units).

The proposed DES API data model is validated against three different real use case scenarios, placing emphasis on the various limitations they posed on the DES data exchange and on the API data model. The solution proposed for one pilot case is detailed in full with different limitations posed by the different specifications and pilot cases specified. Various expansions or enhancements that can be implemented are pointed out in the conclusions.

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken over directly or indirectly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other Irish or foreign examination board.

The thesis work was conducted from 2013 to 2015 under the supervision of Prof. Cathal Heavey at University of Limerick.

Ioannis Papagiannopoulos

Limerick, 2015

Acknowledgements

The current thesis was fully funded from the European Union Seventh Framework Programme (FP7-2012-NMP-ICT-FoF) under grant agreement nº 314364.

First of all, I would like to thank my supervisor Prof. Cathal Heavey. His clear scientific insight and calm approach to research makes for a great supervisor. His guidance and support have been crucial for the successful delivery of this work.

I would also like to express my gratitude to all the members of DREAM project for their assistance and fruitful collaboration throughout this work.

I also wish to thank all the colleagues from the Enterprise Research Centre, and the Department of Design and Manufacturing Technology, University of Limerick. Special thanks to the really good colleagues and friends, George and Panos.

Finally, I would like to thank my parents and siblings for their support.

I would like to dedicate this work to my family; Dimitris,
Konstantina, Despoina, Nikos, Vasiliki.

Contents

List of Tables

LIST OF TABLES

List of Figures

LIST OF FIGURES

Listings

Glossary

ALPS A Language for Process Specification

API Application Programming Interface

ASN.1 Abstract Syntax Notation One

AutomationML Automation Markup Language

B2MML Business To Manufacturing Markup Language

BEA Boundary Element Analysis

BiM Binary MPEG

BOM Base Object Model

BPEL4WS Business Process Execution Language for Web Services

BSON Binary JSON

CAD Computer Aided Design

CAEX Computer Aided Engineering eXchange

CML Chemical Markup Language

CMSD Core Manufacturing Simulation Data

COLLADA COLLAborative Design Activity

COTS Commercial Off-The-Shelf

CPR Core Plan Representation

DB Data Base

DDI Data Documentation Initiative

DDML Document Definition Markup Language

DES Discrete Event Simulation

DITA Darwin Information Typing Architecture

DREAM simulation based application Decision support in Real-time for Efficient Agile Manufacturing

DS-XML Design Extensible Markup Language

EML Election Markup Language

EPPML Extensible Postal Product Model and Language

ERP Enterprise Resource Planning

EXI Efficient XML Interchange

FDMES Flight Dynamic Model Exchange Standard

FEA Finite Element Analysis

FIXML Financial Information Exchange Markup Language

FMI Functional Mock-up Interface

FpML Financial products Markup Language

GUI Graphical User Interface

HR-XML Human Resources XML

HTML5 HyperText Markup Language

HTTP HyperText Transfer Protocol

IDEF Integrated DEFinition Methods

GLOSSARY

IFC Industry Foundation Classes

IFC-XML Industry Foundation Classes XML

IFCXML Industry Foundation Classes XML

IGES Initial Graphics Exchange Specification

JSON JavaScript Object Notation

JSON-LD JSON for Linking Data

KE tool Knowledge Extraction tool

KML Keyhole Markup Language

LML Lightweight Markup Languages

ManPy Manufacturing in Python

MathML Mathematical Markup Language

MEI Music Encoding Initiative

MENOTA Medieval Nordic Text Archive

MES Manufacturing Execution System

MRP Material Requirement Planning

MSML Molecular Simulation Markup Language

NESIS NEutral SImulation Schema

NFS Network File System

NIST National Institute of Standards and Technology

NLM DTD National Library of Medicine Document Type Definition

NoSQL non SQL

OGDL Ordered Graph Data Language

ONIX ONline Information eXchange

ONX Open Node syntaX

OpenDDL Open Data Description Language

OpenMI Open Modelling Interface

OpenSML Open source Simulation Modeling Language

OSCRE Open Standards Consortium for Real Estate

OWL Web Ontology Language

PCXML PopChart XML

PDG Product Development Group

PLM Product Lifecycle Management

PRISM Publishing Requirements for Industry Standard Metadata

PSL Process Specification Language

RDF Resource Description Framework

RSS Rich Site Summary

SBML Systems Biology Markup Language

SDL Simple Declarative Language

SDM Shop Data Model

SDMX-ML Statistical Data and Metadata eXchange Markup Language

SDX Simulation Data eXchange

SDX Simulation Data eXchange

SE Simulation Engine

SEDML Simulation Experiment Description Markup Language

SGML Standard Markup Language

SimPy Simulation in Python

SME Small and Medium Enterprises

SOX Simple Outline XML

SQL Structured Query Language

SRML Simulation Reference Markup Language

SSTML Surgical Simulation and Training Markup Language

STEP STandard for the Exchange of Product model data

StratML Strategy Markup Language

SVG Scalable Vector Graphics

TEI Text Encoding Initiative

TM-XML Trade Mark XML

TOML Tom's Own Markup Language

UBL Universal Business Language

UML Unified Modelling Language

WBF World Batch Forum

XAML Extensible Application Markup Language

XBEL XML Bookmark Exchange Language

XBRL eXtensible Business Reporting Language

XDR XML-Data Reduced

XDR eXternal Data Representation

XSPF XML Shareable Playlist Format

XUL XML User Interface Language

YAML Yaml Ain't Markup Language

List of Publications

1. Dagkakis, G., I. Papagiannopoulos and C. Heavey (2015). "ManPy: an Open Source Software Tool for Building Discrete Event Simulation Models of Manufacturing Systems." Journal of Software: Practice and Experience, John Wiley & Sons, doi: 10.1002/spe.2347.

2. Dagkakis, G., A. Rotondo, I. Papagiannopoulos, C. Heavey, J. Geraghty, P. Young and R. Holland (2014). "From COTS Simulation Software to an Open-source Platform: A Use Case in the Medical Device Industry." Procedia CIRP 25: 283-292.

3. Heavey, C., G. Dagkakis, P. Barlas, I. Papagiannopoulos, S. Robin, M. Mariani and J. Perrin (2014). Development of an open-source discrete event simulation cloud enabled platform. Proceedings of the 2014 Winter Simulation Conference, IEEE Press.

4. Dagkakis, G., I. Papagiannopoulos and C. Heavey (2014). Extending an open-source discrete event simulation platform with new objects. 2014 Operational Research Society Simulation Workshop, SW 2014.

List Of Publications

1

Introduction

1.1 Discrete Event Simulation

The current research deals with simulation and more specifically with Discrete Event Simulation (DES). But what is simulation? Simulation is the imitation of the operation of a real-world process or system over time. As we will deal with computer simulation, let's see what a computer simulation is. In computer science, computer simulation is the technique of representing the real world by a computer program; a simulation should imitate the internal processes and not merely the results of the system being simulated. In that case, a simulation is run on a single computer, or a network of computers, to reproduce the behavior of a system.

Simulation is quite a general term and therefore there is plenty room for confusion. There exist many different tools and packages that allow an end user to perform simulations of a great variety of systems such as physical, biological, medical, computer networks, etc. One can find dedicated simulation tools for e.g. Finite Element Analysis (FEA), Boundary Element Analysis (BEA), ergonomics and many more.

Before defining what Discrete Event Simulation is, let us see what a model is. Model according to the Oxford English Dictionary is "a simplified or idealized description of a system, situation, or process, often in mathematical terms, devised to facilitate calculations and predictions".

1. INTRODUCTION

Mathematical models are the most common models. So called analytical solutions are exact solutions to mathematical models. However, the majority of the real world systems are way to complex to be modelled by means of mathematical equations. The solution of such analytical models is restricted by limited computational resources and the complexity of the systems being modelled. Simulation is used to study models of complex systems; instances of complex models are numerically modelled for different inputs in order to see how output key parameters are affected (Randell 2002).

Three classification axes can be identified according to Banks et al (2010) and Law & Kelton (1991):

- *time axis*: these can be static or dynamic models.
- *probability axis*: these can be deterministic or stochastic models.
- *continuity axis*: these can be discrete or continuous models.

Posing the question this time differently, focusing on the type of simulation we are going to deal with throughout this manuscript: "What is DES?" Contrary to the classification of simulation models defined by Banks et al (2010), Nance (1993) defines three types of simulation; discrete event, continuous, and Monte Carlo. In more detail, they can be articulated as following:

- *Continuous simulation* uses equational models, often of physical systems, which do not portray precise time and state relationships that result in discontinuities. The object of studies using such models do not require the explicit representation of state and time relationships. Examples of such systems are found in ecological modeling, ballistic re-entry, large scale economic models.

- *Monte Carlo simulation* owes its name to its intrinsic gambling similarity, and utilizes models of uncertainty where representation of time is unnecessary. The term is originally attributed to "a situation in which a difficult non-probabilistic problem is solved through the invention of a stochastic process that satisfies the relations of the deterministic problem". Otherwise, it can be described as "the method of repetitive trials". A typical use of Monte Carlo simulation is the approximation of a definite integral by

circumscribing the region with a known geometric shape, then generating random points to estimate the area of the region through the proportion of points falling within the region boundaries.

- *Discrete Event Simulation* utilizes a mathematical/logical model of a physical system that portrays state changes at precise points in simulation time. Both the nature of the state change and the time at which the change occurs mandate precise description. Customers waiting for service, the management of parts of inventory or industrial production lines are typical domains of discrete event simulation.

Discrete Event Simulation (DES) has proved itself to be a powerful tool as the result of the successful implementation in different applications including evaluating potential financial investments, and modeling procedures and processes in various industries, such as manufacturing and health-care. DES can be used to study and compare alternative designs or to solve problems on existing systems (Fowler et al. 2004).

In the current work, we will be dealing with DES problems, and more specifically with the problem of DES data exchange.

1.2 Manufacturing Processes Representation

A fundamental requirement for executing DES is incorporating a data structure that represents process, product and resource information, as well as their interrelations (Shariatzadeh et al. 2012). The required data for DES modelling are heterogeneous and reside in different applications and data-sources. For instance, failure related data may be stored in the databases of maintenance systems, processing times related info may be stored in MES systems and so on. Hence, the representation of this info in a system-neutral format renders the integration of data easier.

A rich list of ontologies that represent manufacturing processes can be found in the literature. This list includes the Process Specification Language (PSL) (Schlenoff et al. 2000), A Language for Process Specification (ALPS) (Catron and Ray 1991), the Core Plan Representation (CPR) (Pease and Carrico 1998),

the <I-N-OVA> constraint model (Tate 1995), the ADACOR Manufacturing Ontology (Borgo and Leitao 2004), Petri Nets (Ochmanska 2004), CMSD information model based on Unified modelling language (UML) (Bergmann et al. 2011; Leong et al. 2006) and several other models. The aim of ontologies is to represent knowledge of a domain, integrating a specific ontology with other types of information.

Graphical representations support the visualization of models for a certain segment of users to whom a "picture is worth a thousand words" (Lacy 2006). As identified by Lacy (2006), graphical representation involves associating icons with statement types and representing control flows with arcs and nodes. Graphical representations are closer to the mental models of the modellers. Graphical representation provide a rich syntax for visually defining concepts such as links, flows, and direction. The graphical representations are serialized into text and used to interchange models as data-files.

1.3 DES data exchange

Developing means for the effective exchange of data between simulations and other manufacturing applications is a critical problem that concerns the scientific community. According to Gang et al (2010), the essential for the emerging requirement of complex system modelling and simulation, is to build interoperable and reusable models and simulations effectively, credibly and rapidly.

Creating tailored data exchange interfaces is too expensive and makes simulation technology unaffordable for most users. The development of reusable, neutral interfaces would make it simpler and could assist the reduction of the cost to integrate simulation and other manufacturing applications. The development of simulation technology and supporting interface standards has been identified repeatedly by industry as top research priority that promises high payback (McLean et al. 2003).

Manufacturing organizations, procedures, and data are growing and becoming more complicated (Ramrez and Nembhard 2004). Manufacturing engineering, product design, production and industrial management decisions include the consideration of many co-dependent factors and variables, which create many com-

plexities in decision making. The widespread use of simulation technology is hindered in the manufacturing industry due to a number of technical and economic barriers. The cost of applying and using simulation technology is high (Carson II 2005). The cost of integrating simulation systems with other manufacturing applications is even higher (Johansson et al. 2007). The cost of implementing DES is excessive owing to the cost of COTS DES packages; the cost of creating a tool or an interface to achieve the integration of DES packages with the different organization IT systems in order to exchange data used by the simulation applications and the time needed for companies to spend on learning how to use them.

As recognized in the literature (Gang et al. 2010; Robinson et al. 2004), complexity, interoperability, and re-usability are major difficulties or problems usually encountered in modelling and simulation. For simple systems, it is usually sufficient to use one or several models to represent the system to be modelled. On the contrary, for complex systems such as building manufacturing shop-floors, building the corresponding simulation systems means to construct various different models and make them run in a complementary fashion. Even with the help of distributed simulation technology, it is still difficult to build models rapidly and make them run with full interoperability (Gang et al. 2010).

Large scale organizations continuously record raw data, and are therefore able to collect large amounts of resource event information. However, usually it is difficult to extract and reuse data for future DES projects (Leong et al. 2006). The difficulty is based on the sharing between data files and simulation models. There is always a critical need to extract, analyse and input data from an organization's Enterprise Resource Planning (ERP) system, Manufacturing Execution Systems (MES), Material Requirement Planning (MRP) systems or other data sources to the simulation model. The development of reusable, neutral standard interfaces would facilitate to reduce the costs associated with simulation model development and enable data exchange between simulation and other manufacturing applications (Johansson et al. 2007).

In the literature, one can find different standards that aim to address the problem of DES data exchange. The long list includes CMSD, NIST SDM, SDX, AutomationML, STEP-ISO 10303 (Boulonne et al. 2010; Harward and Harrell 2006; Hossain et al. 2012; Shariatzadeh et al. 2012; Sly and Moorthy 2001). Almost

all these proposed data exchange formats are expressed in eXtensible Markup Language (XML). Otherwise, the files containing the DES data are written in simpler file formats that offer no schema description capabilities, i.e. Simulation Data eXchange (SDX) files (Burnett et al. 2008). The choice of XML seemed logical in the past as the most widespread data exchange format which was platform independent and widely accepted by almost all software packages (McLean et al. 2003; Rohl and Uhrmacher 2005). It also offered the capability of data structure definition via the use of XML schemas. Nowadays, there exist numerous data exchange formats with similar capabilities which can replace XML as DES data exchange formats. For example, JavaScript Object Notation (JSON `http://json.org/`), Yml Ain't Markup Language (YAML `http://yaml.org/`) and many others.

1.4 Problem Statement

The study presented here is conducted as part of a European FP7 project called DREAM ("simulation based application Decision support in Real-time for Efficient Agile Manufacturing", `http://dream-simulation.eu`) project. The scope of DREAM project is to increase the competitiveness of the European manufacturing sector through targeting the advancement of DES technology beyond the current state of the art. The main goal of the research is the development of a simulation platform based on a new open-source, expandable and semantic free Simulation Engine (SE). The newly developed SE is called ManPy (Manufacturing in Python). The SE co-operates with other modules such as Graphical User Interface (GUI) developed in JavaScript, and a Knowledge Extraction (KE) tool. All the modules are structured to be independent but co-operating in the final version of the platform.

Within the scope of DREAM, the issue of data sharing between manufacturing and simulation applications has to be addressed. DES data is to be collected from different modules of the platform as well as from external manufacturing applications. The technologies to be used for the development the interface must guarantee the interoperability between the different modules of the platform. It is requested that they are open-source, easy to learn by parties not familiar with

the nomenclature of DES, and easy to adopt to the specific needs of the DES modelling. It is desirable that the Application Programming Interface (API) to be used will be expendable and easily configurable so that the platform can be modified and the different modules synchronized only by the use of the API.

The API to be implemented should consider a short learning curve for the developer and the end-user or modeller. Given the fact that the developer parties involved in the development of the platform have different backgrounds, a simple and intuitive way to exchange DES data is required in order to enhance the adoption of the API by third party partners not accustomed to DES specificities.

Last but not least, it is required that the DES model description to be implemented by the data exchange API resembles the graphical representations of simulation models used by commercial simulation software (Simul8 - `http://www.simul8.com`, Siemens Plant Simulation - `http://tinyurl.com/nv8vnba`). It is common practice among COTS simulation tools, to address the problem of designing a model by describing the flow logic as a network of nodes and edges. Modellers are familiar with that way of designing models and it is considered good practice for explaining DES terminology to beginners as third party developers.

1.5 Research Objectives

In an attempt to address the issues of data sharing between manufacturing and simulation applications, a neutral, reusable JSON based API used to exchange DES data between different simulation and other manufacturing applications will be presented. To facilitate the communication between the different DREAM modules, a semantic free, expandable and neutral reusable data exchange interface is required. Within the scope of the research project, the data exchange is imperative to be implemented in a fast and intuitive way. The requirements are to a great extend specified by the technologies used for the development of the platform.

One of the main considerations at the early stages of the research was if one of the existing standards should be deployed for the communication between the DREAM modules or a new one should be developed to better address our needs. A literature review of the existing standards used in DES data exchange

was conducted for that purpose, and will be presented in detail in the following chapter.

The need for neutral data exchange formats has been addressed by different standards. Early in the lifetime of the project, it was noted that there exist no DES data format based on JavaScript Object Notation (JSON), while there is a plethora of solutions based on eXtensible Markup Language (XML). This luck of JSON based data exchange format focused on DES, together with the need to implement an API that is inherently related with the tools used for the development of the different modules of the platform (JavaScript, SimPy-Python), defined the objectives of the present research. Lastly, the requirement of a end-user friendly API capable of describing DES models' topologies as a network of nodes and edges.

In order to address the challenges identified, this work focuses on the following issues in the context of developing an API which:

- is based on a format that is inherently related to the technologies used for the development of different modules of the platform, namely JavaScript and Python.
- is expandable and configurable so that it can enable the synchronization of the different modules of the platform in an easy manner.
- is neutral so that
  - it can be adopted and used by other platforms.
  - the platform can integrate any external module or manufacturing application.
- can or is described by a schema, so that it can be easily documented, validated, and disseminated.
- can be directly integrated with a graph editor for designing DES model topologies as a directed graph of nodes and edges.

1.6 Thesis Outline

The remaining chapters of this dissertation are as follows:

Chapter 2 is an introductory presentation of the different types of data exchange formats available. More specifically, data exchange formats used in DES are discussed in detail. Initially, different data exchange formats are charted in order to give an overview of the available technologies for creating interfaces. Different representations used by the DES data exchange standards are listed. The driving need behind the proposed approach is presented. Taking the discussion on step further, various business and manufacturing representation methods are discussed. The ontologies used for representing such processes are mapped directly to data structures.

Chapter 3 presents the Simulation Platform developed within DREAM project. The major objectives driving the current research are discussed. All the different modules of the platform are structured to be independent but cooperating. The need for interface addressing that need is discussed.

Chapter 4 presents the DES Application Programming Interface (API) data model proposed. The different elements of the data model are detailed by means of examples. The information flow through the platform is discussed with regards the structuring and the use of the DES API data model. The elements that are used in the DES data exchange are distinguished from the elements used for the configuration and synchronization of the platform.

Chapter 5 describes the different pilot cases that were studied and used for the evaluation of the DES API introduced in this work. More detailed information is provided about the set-up, the configuration, and implementation of an DES API instance for one of the use case scenarios. Examples of information parsing and analysing are given through the introduction of the corresponding plugins.

Chapter 6 draws conclusions and evaluates the proposed solution to the problem of interfacing different manufacturing and discrete event simulation modules. The innovations introduced are outlined. Lastly, it suggests possible future works.

A series of documents have been included in the Appendix section of this dissertation. These are:

- *Appendix A* provides a short list of lightweight markup languages that are used for presentation rather than data exchange purposes.

- *Appendix B* presents the general simulation properties of pilot case described in Chapter 5, as defined by the pilot case's DES API instance.
- *Appendix C* includes detailed examples of processing/pre-processing/post-processing, and support plugins used for the parsing, analysis, and evaluation of the DES related information before/after they are processed by the DES engine.

Attached to this dissertation is a CD containing the following items:

- *Appendix D*: A fully featured instance of the JSON DES API data file. The file contains all the data required for the configuration and instantiation of the platform, the DES related input data, the DES model, and the returned from the DES engine results after a simulation run.

2

Literature Review

In the present chapter, we are going to take a closer look at the options offered for creating an Application Programming Interfaces (APIs) using one of the available data exchange formats. An effort is made to chart the available formats for data interchange, ranging from those that are already used for decades to the most recent innovations, and from the most generic ones to tailored solutions intended to address very specific needs.

As a next step, a literature research is conducted on the available manufacturing or business data exchange standards used in the field of DES. There are numerous solution approaches to the problem of collecting and exchanging data for DES purposes. There exist solutions that are specifically intended to deal with DES needs, and others that initially where tailored to address manufacturing or business processes related problems.

A short review on the different manufacturing processes representations that can be found in the literature, is conducted. Every data exchange standard that aims to address the problem of describing manufacturing processes and the information flow within them, requires an ontology that maps the logic of these processes.

The reasoning behind the solution proposed in the current work is finally explained. A new simple representation of the manufacturing processes is proposed. It is based on the directional graph approach used by many Commercial Of The Self (COTS) DES software suits. The logic for choosing the JSON data exchange format is also explained.

2.1 Data Exchange Formats

According to Fagin et al (2005), data exchange is:

> the process of taking data structured under a source schema and actually transforming it into data structured under a target schema, so that the target data is an accurate representation of the source data.

Data exchange is similar to the related concept of data integration except that data is actually restructured (with possible loss of content) in data exchange. Trying to identify the origin of data exchange and the relative formats, we find out that the driving need is the request for formats that serialize data. Such need exists since the appearance of computer networks. But what types of data exchange protocols are there.

Initially the problem was addressed by binary protocols. These protocols contain data which were not human readable. Custom-defined ad-hoc schemas were used. Those schemas had to be agreed upon by the sender and the receiver. Eventually, those schemas gave way to standards as ASN.1 and XDR.

ASN.1 stands for Abstract Syntax Notation One (Information processing systems – Open Systems Interconnection 1987) and originated somewhere in 1984. ASN.1 is a standard and notation that describes the rules and structures for representing, encoding, transmitting, and decoding data in computer networking. The rules that constitute the standard enable the representation of data structures/objects that are independent of machine/network/language specific encoding techniques. The formal notation used renders it possible to automate the task of data validation. ASN.1 defines the abstract syntax of information but does not restrict the way the information is encoded. Various encoding rules provide the transfer syntax of the data values whose abstract syntax is described in ASN.1. Its abstract syntax together with specific ASN.1 encoding rules facilitates the exchange of structured data especially between application programs over networks by describing data structures in a way that is independent of machine architecture and implementation language.

XDR stands for eXternal Data Representation and was really successful during the mid 1980s (Srinivasan 1995). It is still used to serialize compiled JavaScript. XDR is a standard data serialization format for uses such as computer network protocols. It allows data to be transferred between different kinds of computer systems and it is independent of the transport layer. Various data types are supported by XDR. It is used by many systems such as the Network file system (NFS), the ZFS file system, the Network Data Management Protocol, etc.

2.1.1 Human readable formats

As the volume of data that were requested to be transmitted grew bigger, new standards were required. With the emergence of the Hyper Text Markup Language (HTML) and the Standard Generalized Markup Language (SGML) (Raggett et al. 1999), a human readable data exchange format was thought to be a good idea. The eXtensible Markup Language (XML) was one of the first to address that problem, and became widely used. With the growth and the diversity of the various network applications, other alternatives also emerged, each one with the intention to address different problems that emerged through the use of these applications.

A short review of different Data Exchange Formats is performed and the results are presented in the form of a map (see Table 2.1). The main criteria considered for the data exchange formats review were:

- a well-supported ecosystem of parsers and translators so that it can easily be integrated in different platforms.
- Referencing within the data structures of the documents.
- Schema support, as it is crucial for validation of the data transferred, and for the synchronization of the different modules of the platform.

The formats considered are XML, JSON, BSON, YAML, TOML, Protocol Buffers, SDL, ONX, Property lists, Sofu, OGDL, and OpenDDL. Data Exchange Formats that are intended to address needs of specific fields are not considered. The problems that they focus on are not relevant to the main subject of the current study.

From the options provided in Table 2.1, the major candidates are XML, YAML, and JSON. The rest are either immature and thus their ecosystem of

Table 2.1: Data Exchange Formats' attributes

	XML	JSON	BSON	YAML	TOML	PB	SDL	ONX	PL	Sofu	OGDL	OpenDDL
text based	x	x	x	x	x		x	x	x	x	x	x
human readable	x	x	x	x	x		x	x	x	x	x	x
position independent	x	x	x	x	x	x	x	x	x	x	x	x
unconstrained	x	x	x	x	x	x	x	x	x	?[1]	x	x
markup	x							x				
different data structures support	x	x	x	x	x	x	x	x	x	x	x	x
referencing	x	x		x			x			?[1]	x	x
version-less	x										?[1]	?[1]
schema support	x	x[2]	x[3]	x[2]		x[4]				?[1]	x	x[2]
binary support	x		x			x	x		x	?[1]	x	

schemas and parsers are not as developed as for the three major competitors, or they lack important features such as schemas or referencing capabilities. In the following subsections, we will discuss the different data exchange formats that were reviewed, and eventually the result of the evaluation will be presented at the end of the chapter.

2.1.1.1 XML

The Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine readable (Bray et al. 2006). The design goals of XML emphasize simplicity, generality and usability across the Internet. It is used for the repre-

[1]Not identified.
[2]External schema descriptors, can be optionally schema-less.
[3]BSON is derived from JSON, JSON-schemas can be applied.
[4]A definition file can be used as external schema.

sentation of arbitrary data structures such as those used in web services. XML technologies or in other words the ecology of XML modules provide useful services that enable the use of XML as text mark-up, transform XML docs to any other type of document recognized by a browser, navigate through an XML document, create links and reference between data structures and resources. An example of XML code can be found in listing 2.1.

Listing 2.1: XML example

```
1       <menu id="file" value="File">
2         <popup>
3           <menuitem value="New" onclick="CreateNewDoc()" />
4           <menuitem value="Open" onclick="OpenDoc()" />
5           <menuitem value="Close" onclick="CloseDoc()" />
6         </popup>
7       </menu>
```

Numerous document formats have been developed using XML syntax, naming RSS (RSS Advisory Board 2009), Atom (Nottingham and Sayre 2005), SOAP (Gudgin et al. 2005), and XHTML (Axelsson et al. 2010) as examples. In addition, several schema systems exists to aid in the definition of XML-based languages, while many APIs have been developed to aid the processing of XML data. Different Data Exchange formats that are used in diverse fields are based on XML. There exists a corresponding XML schema for every Data Exchange format based on XML. A short list of XML schemas with regards to the purpose of use is given in table 2.2.

XML despite the multiple features tha offers to users, still has complex schemas. XML performance is not one of the virtues of the language. Significant processing overhead is imposed due to the formatting of the XML documents. An attempt to reduce the overhead due to tags that are used by XML, is SOX. SOX stands for Simple Outline XML. SOX is a compressed way of writing XML. It uses indenting to represent the structure of an XML document, eliminating the need for the closing tags.

An other compact representation of XML is Binary XML. It generally reduces the verbosity of XML documents but hinders the use of ordinary text editors for viewing and editing. It replaces the standard XML format where the performance

Table 2.2: XML schemas

Field	Schemas
Bookmarks	XBEL
Brewing	BeerXML
Business	XBRL, PCXML, eCOM XML, UBL, HR-XML StratML, OSCRE, SRML
Elections	EML
Engineering	Proteus Schema, IFC-XML
Financial	FpML, FIXML
Geotagging	KML
GUIs	JAXFront, GLADE, Kparts, XUL, Xforms, XAML
Humanities texts	TEI, EpiDoc, MENOTA, MEI
Industrial property	TM-XML, DS-XML
Manufacturing	AutomationML, B2MML
Math and Science	MathML, N42
Metadata	RDF, ONIX, DDML, PRISM
Music Playlists	XSPF
News syndication	Atom, RSS
Paper/forest products	papiNet, EPPML
Publishing	NLM DTD, DITA, DocBook, PRISM
Statistics	SDMX-ML, DDI
Simulation	BOM, CMSD
Vector Images	SVG

is insufficient. The cost of parsing is also reduced. Additionally, random access and indexing of XML documents is rendered possible. The most prominent effort to encode XML documents in binary format is the Efficient XML Interchange (EXI) (Schneider et al. 2014). There exits also a Binary MPEG format for XML (BiM) which has been standardized and is used in many standards for digital and mobile TV (DeCuetos et al. 2006).

2.1.1.2 JSON

JSON is a smaller alternative to XML which emerged following the JavaScript widespread adoption over the Internet. JSON stands for JavaScript Object Notation (ECMA International 2013). According to Crockford (2006), JSON is a lightweight, text based, language-independent data interchange format, it was derived from the ECMAScript Programming Language Standard. JSON defines a small set of formatting rules for the portable representation of structured data. In short, JSON is a text format for serialization of structured data (listing 2.2).

Listing 2.2: JSON example

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
   "menuitem": [
     {"value": "New", "onclick": "CreateNewDoc()"},
     {"value": "Open", "onclick": "OpenDoc()"},
     {"value": "Close", "onclick": "CloseDoc()"}
   ]
  }
}}
```

JSON has several interesting benefits which are really worth mentioning. It is native with JavaScript and Python and all the JSON objects are directly translated to JavaScript objects or Python dictionaries. Actually, JSON is read directly as JavaScript. Due to its wide adoption, it is supported by various tools and libraries. Its grammar is really simplistic and can be described by only 5 diagrams (ECMA International 2013). Thus, its adoption is straightforward, and can directly be used by developers with no prior experience.

Furthermore, JSON has the data structure of the Web. It is a simple data format that allows programmers to store and communicate sets of values, lists, and key-value mapping across systems. Last but not least, it is really important to mention that it has an expandable and well structured schema (Galiegue and Court 2013). The specifications are available on `http://json-schema.org/`. This renders the JSON documents really agile and easy to change/expand. A selection of JSON schemas can be found at `http://schemastore.org/`.

The use of JSON has recently expanded to new areas, such as databases (Chasseur et al. 2013). It actually gave rise to NoSQL document databases. Numerous NoSQL databases have been developed, catering in particular to developers who favour the JSON format.

On the contrary, JSON lucks several desirable virtues. It has no support for date data type and it does not support comments. Several variants emerged in order to compensate for the shortcomings of JSON. BSON which stands for binary JSON, is used in MongoDB (a NoSQL database) as an alternative to JSON. JSON-LD is JSON based format for serializing Linked Data.

Markup languages have been developed for the purposed of mapping between XML and JSON. JSON Markup language (JSONML) is a lightweight markup language used to map between XML and JSON (`http://www.jsonml.org/`). Its introduction aimed to achieve a compact format for transporting XML-based markup as JSON which allows it to be lossless converted back to its original format. It can be characterized as presentation oriented language.

2.1.1.3 YAML

The Yaml Aint Markup Language (YAML) is a human readable data serialization format for all programming languages (BenKiki et al. 2009). It takes concepts from programming languages such as C, Python, and ideas from XML and the data format of electronic mail (RFC 2822).

YAML syntax was designed to be easily mapped to data types common to most high-level language, namely: list, associative array, and scalar. It is indented outline makes it especially well suited for tasks where humans are likely to check

the data structures, i.e. configuration files, dumping when debugging, document headers etc. Full specification is available at `http://yaml.org/`.

YAML has several remarkable advantages. Even though it has a lengthy documentation for a standard, it is concise and easily readable, as it borrows from python the use of whitespace, and avoids quotation marks, brackets, braces or open/close-tags. It also contains references, as ways to refer to previously defined elements. Furthermore, YAML supports all the standard data types and can map easily to lists, hashes, or individual data items. In addition, it has a compact syntax for relational data.

YAML shares similarities with JSON, XML, and SDL (Simple Declarative Language). JSON syntax is the basis for YAML and actually YAML is a superset of JSON. Most JSON documents can be parsed by YAML parsers, as for example yamljs available at `https://www.npmjs.com/package/yamljs`. JSON semantic structure is equivalent to the optional 'in-line-style' of writing YAML. Going one step further though, YAML introduces comments, extensible data types, relational anchors, and strings without quotation marks. An example YAML can be seen in listing 2.3.

Listing 2.3: YAML example

```yaml
---
menu:
  id: "file"
  value: "File"
  popup:
    menuitem:
      -
        value: "New"
        onclick: "CreateNewDoc()"
      -
        value: "Open"
        onclick: "OpenDoc()"
      -
        value: "Close"
        onclick: "CloseDoc()"
```

Comparing with XML, YAML is lacking the notion of tag attributes, but instead has extensible type declarations. It doesn't have a language-defined document schema and thus self-validation of documents is not possible. Though, it is possible to use external schema descriptors. YAXML represent YAML structures in XML and allows XML schemas to be applied to YAML.

YAML offers a simple relational scheme that allows repetitions of identical data to be referenced from multiple points in the tree than entered redundantly at these points. In other languages in the likes of SDL and JSON, data are represented only in a hierarchical model where each child can have only a single parent. The parser eventually expands these references into the fully populated data structures. Unlike YAML, XML processors do not expand references.

2.1.1.4 Other candidates

BSON Binary JSON (BSON) is a computer data interchange format used mainly as data storage and network transfer in the MongoDB database. The specification is available at `http://bsonspec.org/`. As mentioned in Section 2.1.1.2, NoSQL databases as MongoDB resulted from the widespread use of JSON. Hence, it is easy to deduce that BSON originates in JSON and extends it.

Like JSON, BSON supports the embedding of documents and arrays within other documents and arrays. Extending JSON, it also contains extensions that allow representation of data types that are not part of the JSON specification, such as date and Binary Data type.

BSON contains a lot more of metadata compared to JSON. This metadata is mostly length prefixes so that you can skip through data that may be of no interest. The choice is dependent on the specific needs of the applications. Storing enormous documents when the storage space is limited, use of JSON is suggested because it is more compact and space efficient. On the other hand, storing document while reducing wait time is of highest priority than saving some disk space, BSON is a better option.

It can be compared to other binary interchange formats, as Protocol Buffers (see Section 2.1.1.4, paragraph on Protocol Buffers). It is more schema-less than Protocol buffers, which means that it is more flexible, but it has an overhead due to field names within the serialized data, which impacts negatively the efficiency.

TOML TOML stands for Tom's Obvious Minimal Language (PrestonWerner 2015). It aims to be a minimal configuration file format easy to read due to obvious semantics. Furthermore, it should be easy to parse into data structures in a wide variety of languages. Though, it is not yet used as common alternative to the other more widely accepted data exchange formats.

Listing 2.4: TOML example

```
[menu]
  id = "file"
  value = "File"
  [[menu.popup.menuitem]]
    value = "New"
    onclick = "CreateNewDoc()"
  [[menu.popup.menuitem]]
    value = "Open"
    onclick = "OpenDoc()"
  [[menu.popup.menuitem]]
    value = "Close"
    onclick = "CloseDoc()"
```

TOML is made after the format of .ini files, incorporating many conveniences. It is similar to configuration files with key-value pairs grouped by bracketed item names. It offers a good arsenal of advantages. For example, it has the brevity of JSON, using a different notational schema. In addition, it has obvious semantics, and it is taken-off the widely used .ini files format. A sample snippet of a TOML file can be seen in listing 2.4. Most probably the list will not be parsed correctly as the feature is not yet fully supported.

Protocol buffers Protocol buffers (PB) is a method of serializing structured data (Carey 2008). The main goal of the format is simplicity and performance. They are used in the scope of developing programs to communicate with each other. It is also used for storing data within the same scope.

The method involves the use of an interface description language that describes the structure of some data, and a program that generates source code

from that description for generating or parsing a stream of bytes that represents the structured data. Data structures that are called messages, and services are defined in a definition file. It can be said that the definition file specifies a schema for the data. According the documentation, new fields and properties can be easily introduced, and intermediate servers that didn't need to inspect the data could simply parse it and pass through the data without needing to know about all the fields.

The compilation of the definition file generates code that can be invoked by a sender or recipient of these data structures. Protocol Buffers are serialized into a binary wire format, forwards/backwards-compatible, but not self-describing. There is no defined way to include or refer to an external definition such as a schema, within the Protocol Buffer file.

Protocol buffers are designed to be smaller and faster than XML. They are considered to be less ambiguous and generate data access classes that are easier to use pragmatically.

SDL SDL stands for Simple Declarative Language and provides an easy way to describe lists, maps, and trees of typed data in a compact, easy to read representation (Leuck 2012). The simple and intuitive API allows you to read, write and access all the data-structures using a single class.

For property files, configuration files, logs, and simple serialization requirements, SDL provides an alternative to XML. Contrary to XML, SDL is particularly suited expressing basic data structures. There are implementations available in in Java, .NET. Ports for C++ and other languages were planned for the near future, at the time the review was performed.

ONX Open Node syntaX (ONX) was created as an alternative to the markup languages that were used at the time and is not supported any more Jacobs 2005. The documentation was last updated in 2005. It was designed to be data-oriented instead of document-oriented. It was intended to cover the need for platform-independent data transfer over distributed systems.

ONX was organized in information blocks were nodes accompanied by their contents. Within each node, there might be zero or additional nodes. The contents could be nodes-container nodes, or values-value nodes.

ONX had several interesting features. It had a minimal set of rules in the likes of JSON. It could be described as generic and thus applicable in a wide variety of applications. It was compact so that the overall size of data to be transferred was kept minimal. Finally, simple parser implementations were meant to be supported allowing bug-free and speed implementation of parsers.

Property lists A Property list (PL) is a structured data representation and is used as convenient way to store, organize, and access standard types of data (Apple Inc. 2010). We can claim that property lists are used to store serialized objects. The are often used to store users' settings. They can also contain information about bundles and applications.

They are extensively used by applications on OS X and iOS. They are based on an abstraction for expressing simple hierarchies of data. The data items in a property list are of a limited number of types. They can be primitive types or containers of values. Primitives types can be strings, numbers, binary data, dates, and boolean type. Containers can contain other containers as well as other primitives. Hence, a root property list is at the top of this hierarchy and might be containing other arrays and dictionaries, as well as primitive types.

Sofu Sofu is a file format that is designed to store data in a form that both humans and computer programs can easily read and write. The detailed specification is available at `http://sofu.sourceforge.net/`. It is based on plain text, meaning that Sofu files can be edited with any text editor. It is quite immature as a file format as its current version is 0.2.

The 'o' in sofu stands for Object-based. It has a tree layout which fits in perfectly with Object Oriented Programming. A Sofu file is made up of three different kinds of objects: values, lists and maps.

OGDL Ordered Graph Data Language (OGDL) is a structured textual format that represents information in the form of graphs, where the nodes are strings and

the arcs or edges are spaces or indentation. OGDL includes a schema notation and path traversal notation. There is also a binary representation. Extended information can be found at `http://ogdl.org/`.

In the case of OGDL, schema is a definition of a type. Some basic types are already defined, and schemas are just complex types. The specification though is yet immature. The use of schemas has yet to be proved useful.

OpenDDL The Open Data Description Language (OpenDDL) is a generic text-based language that is designed to store arbitrary data in a concise human-readable format (Lengyel 2013). It can be used as a means for easily exchanging information among many programs or simply as a method for storing a program's data in an editable format. OpenDDL is a new Data Description Language and its initial specification was released in 2013. It is based on C and PHP.

One thing that sets OpenDDL apart from other data languages is the fact that each unit of data in an OpenDDL file has an explicitly specified type. This way can protect the integrity of the data. The data structures in an OpenDDL file are organized as a collection of trees. The sequence of structures that compose the OpenDDL files, can be either user-defined types or built-in primitive data types. The language also includes a built-in mechanism for making references from one data structure to any other data structure, allowing in such a way the contents of a file to be structured as a directed graph. Schemas can be 'built upon OpenDDL itself in order to define valid type usages in derivative file formats (Lengyel 2015).

OpenDDL is intended to be used as foundation for higher-level data formats and thus, it is designed with the aim of being minimalistic. It assigns no meaning to any data beyond its hierarchical organization, and it imposes no restrictions on the composition of data structures. Lastly, comments are supported unlike JSON.

2.1.2 Presentation oriented languages

Presentation oriented languages are focused on textual representation and not on data serialization. Most of this formats are lightweight markup languages (LML), also termed simple or humane markup language. Lightweight Markup

Languages (LML) are markup languages with simple, unobtrusive syntax. They are designed to be easily created or edited with any generic text editor, easy to read in its raw form. They are used in applications where it may be necessary to read the raw document as well as the final rendered output. A list of LML examples are presented in appendix A and Table 1. Different approaches using various notations are also available. We can name JsonML as an example. As they are focused on presentation of data, they do not provide the means for data exchange.

2.2 Data Exchange Standards in Simulation

After a simple bibliographic research, numerous standards can be reported to have been used for simulation purposes. The problem of generating an environment where different kinds of software can be meshed seamlessly with each other, has been approached in many different ways. It is almost certain that the data content of any simulation model will eventually be complemented from external sources of information, otherwise the data content of the model will initially be increased (Laasonen 2010). The latter is not always feasible in real time applications.

In different fields, various approaches have been used. In fire, thermal, and structural simulation of buildings, where CAD information is used, STEP (STandard for the Exchange of Product model data) and IFC (Industry Foundation Classes) standard have been used (Dimyadi et al. 2008; Laasonen 2010; Nytsch-Geusen et al. 2003). In building automation, data exchange can be standardized by the use of IGES (Initial Graphics Exchange Specification), STEP, IFC, and CAEX (Computer Aided Engineering eXchange) (Runde and Fay 2008). For evaluation of complex cyber-physical systems, where integration of heterogeneous simulation environments is requested (electrical, thermal, mechanical, cyber), the Function Mock-up Interface has been developed (Neema et al. 2014).

For the representation and exchange of biochemical models in simulations of biochemical systems, SBML has been developed (Hucka et al. 2003). There is also SEDML (Simulation Experiment Description Markup Language) for the encoding and exchange of simulation descriptions on computational models or biological systems (Waltemath et al. 2011).

More standards and languages have been developed for the various simulation problems. Some examples of the variety are listed in Table 2.3. Data exchange formats that can be used in the DES field are discussed in the Section 2.3.

Table 2.3: Simulation Data Representation & Exchange Formats

Language	
IFCXML	Construction industry data (Nisbet and Liebich 2007)
FMI	Cyber-physical systems (Blochwitz et al. 2011)
SEDML	simulation descriptions on computational models or biological systems (Waltemath et al. 2011)
SBML	Computational models for biological processes (Hucka et al. 2003)
FDMES	Flight Dynamic Model Exchange Standard for flight modelling and simulation (Jackson et al. 2011)
MSML	Molecular Simulation Markup Language (Birkenheuer et al. 2012)
CML	Chemical Markup Language (Birkenheuer et al. 2012)
CostGlue	Scientific metadata (Furfari et al. 2008)
OpenMI	Originally for simulation of environmental processes (Gregersen et al. 2007)
SSTML	Surgical Simulation and Training Markup Language (Bacon et al. 2006)

2.3 Data Exchange Standards in DES

As described in the introductory Chapter 1, discrete event simulation and manufacturing data storages can automatically exchange information with means of interoperability (Johansson et al. 2007). To support interoperability, applications must organize and connect information consistently. As expressed by Umeda and Jones (1997), the increasing need to integrate simulation and other manufacturing applications and databases was a product of modellers' requirements;

- it is necessary to integrate simulation software with production planning software so that accurate estimates of delivery times can be provided to customers.
- it is important to link simulation with real time control system.
- integration of simulation and scheduling is desirable.
- link simulation with cost estimating.
- distributed, autonomous simulation will be required in the near future.

Data exchange interfaces and standards are accepted in main areas like process engineering, process control engineering and in manufacturing automation engineering (Boulonne et al. 2010). After a review of the literature we identified several standards that have been applied to exchange data; these are STEP-ISO 10303, Automation Markup Language (Automation ML). ISA-95, NIST SDM, Simulation Data eXchange (SDX), and Core Manufacturing Simulation Data (CMSD). In the next sections we describe the identified data exchange interfaces.

2.3.1 SDX

SDX stands for Simulation Data eXchange and which was developed to address the need for an automated integration of production systems layouts and simulation technologies for manufacturing. It was originally developed as a common data format for generating DES models and 3D model animations directly from CAD drawings (McLean et al. 2003). CAD objects for facility layout design (e.g. conveyors, cranes, tables, etc) are embedded with simulation relevant data such as Time to Fail, Time to Repair, Cycle times, conveyor speed, etc (Sly and Moorthy 2001). Simulation relevant data are exported to a standardized XML text file. This file constitutes the data format called SDX. In other words that file can serve as an input to the automatic generation of DES models. It has a complete set of organized data from both the CAD model as well as from other data sources containing manufacturing related data. A translator is needed to generate automatically the simulation model from the ASCII file (Moorthy 1999). SDX standard was developed and coordinated by the National Institute of Standards and Technology (NIST). SDX can be described as a good first cut at a neutral file format for simulation data.

2.3.2 NIST SDM

SDM is the Shop Data Model developed by the National Institute of Standards and Technology (NIST) as part of the the efforts towards the development of standard data interfaces (McLean et al. 2005). The information model was intended to be used for representing and exchanging machine shop data, between manufacturing execution, scheduling, and simulation systems. An other goal was

to promote the information model as a standard data interface for manufacturing simulators. The plant layout data elements from the SDX effort were used as basis for SDM. It can be described as a neutral file format for the DES of manufacturing systems. The specification of SDM in two formats which are UML and XML (Harward and Harrell 2006). SDM information model was eventually transferred to the Core Manufacturing Simulation Data (CMSD) Product Development Group (PDG).

2.3.3 OpenSML

OpenSML is based focuses on an open-source development approach in order to create a simulation modelling language as a method to provide common core simulation functionality for the DES industry (Kilgore 2001). It was originally developed as a Java-based programming language and aimed to develop a standard DES language that is readable, modular, and extensible. OpenSML was extended by using universal, language independent XML (Wiedemann 2002). In order to make use of the neutral file format, the system architecture consisting of heterogeneous modules, is designed and implemented based on the proposed neutral simulation model for an interoperable environment. The data structure and expression are referred to as PLM services, and the contents are referred to as SDX and NIST SDM.

2.3.4 SRML

Simulation Reference Markup Language (SRML) is an XML-based markup language developed at Boeing for representing simulation models. The creation of simulation models using SRML is patterned after the process of web page authoring. SRML supports item classes which support hierarchical containment relationships, item quantities for describing large nested numerical items, links for use in referencing, external files. This capabilities enabled the running of simulation models in a Web-based environment (Reichenthal 2002). In other words, SRML is an XML application that can describe the behaviour for distributed simulation models, and its runtime environment is software that is capable of

executing those models. An XML-Data Reduced (XDR) schema for SRML on `http://www.w3.org/TR/SRML/`.

2.3.5 NESIS

NESIS stands for NEutral SImulation Schema and is intended to solve the problem of simulation models exchange simulation and other manufacturing applications (Lee et al. 2011). It based on several neutral file formats such as NIST SDM, SDX and PLM Services (Stark 2011). NESIS has been introduced to solve particular problems inherent in the previously mentioned file formats:

- SDX does not consider hierarchical operational structures and thus cannot support Process-BOMs (Bill Of Materials) and additionally is not enough to express all kinds of information of simulation.
- NIST SDM cannot describe sufficiently information related to resources as it is mainly focused on machine operations.
- PLM Services mainly focuses on exchange of product information. Thus, other information, such as process and resource information is insufficient. In any way, PLM Services is defined from the business process' point of view.

NESIS consists of five schemas; one for products, one for process, one for resources, one for configuration and a fifth one for defining the routing information for the simulation.

2.3.6 STEP-ISO 10303

STEP is the acronym for Standard for the Exchange of Product model data and is specified within ISO 10303. STEP has been developed to provide the means for the exchange of product information along the complete product life cycle (Pratt 2001). To address the needs of interoperability, durability and expandability i.e. it should be feasible to swap and share data between applications irrespective of hardware and operating system, the information standard ISO 10303 (STEP) offers means able to describe product data throughout the life-cycle of a product,

independent of any specific system and execution method. Therefore, this specification goes beyond the capability of pure data exchange interfaces (Kjellberg et al. 2009). This standard mainly stems from the needs of mechanical products within automotive industry and is often applied for the representation of information such as geometry and kinematics. The step standard ISO 10303-214 - Core Data for the Automotive Mechanical Design Processes (AP214) has previously been assessed with regard to the use within production systems applications, and for teh handling of process data such as production sequences and resource capabilities (Falkman et al. 2008).

2.3.7 AutomationML

AutomationML is another standard aiming to connect different engineering applications in the design and development of production systems. Using several existing standards, AutomationML consists of information about factory topology, geometry, kinematics, and logics (sequencing, behaviour and control). AutomationML is an Extensible Markup Language (XML) based neutral data format for saving and sharing operating information. The goal is to intersect the varied tool landscape of engineering tools in mechanical plant engineering, electrical design, PLC, and robot control fields (Drath et al. 2008). AutomationML can be applied to demonstrate the workflow of process and its association with products and manufacturing resources in an abstract and tool independent way (Luder et al. 2010). It has specified its top-level format adopting CAEX and defining a suitable CEAX profile. CAEX itself is a meta format based on XML, specially designed for the modeling of hierarchical structures. Another data format is frequently used by this standard called COLLADA, and it is mostly applied for geometry and kinematics data. The standard is divided into three core components for the description of logic data related to particular AutomationML objects or groups of objects. These are the “behaviour” representation as the response of an AutomationML object on external signals, the “sequencing” representation as the internal control performance of an AutomationML object, and the “interlocking” representation allowable or forbidden situations between AutomationML objects (Drath et al. 2008). Basically AutomationML supports the following concepts:

Factory topology - CAEX, geometry - COLLADA, kinematic - COLLADA, behavior description - PLCOpen-XML, and references from CAEX to external data (Drath 2010).

2.3.8 ISA-95

ISA-95, International Standard for the Integration is a standardization of enterprise and control systems, recognized globally as IEC/ISO 62264, offers Unified Modeling Language (UML) models and technology that can be applied to decide which data needs to be shared between systems for sales, finance etc. ISA-95 has been developed for global manufacturers. It was designed to be exploited in all industries, and in all kind of processes, like batch processes, continuous and repetitive processes (Scholten 2007). ISA-95 introduced by ISA (International Society of Automation) is the international standard for the integration of enterprise and control systems. The goals of ISA-95 are to offer coherent terminology that is a foundation for supplier and manufacturer connections, offer coherent information models, and to offer coherent operations models which is a basis for clarifying application functionality and how information is to be utilised (Gifford 2007). As an overall instruction, ISA-95 defines all trade from ERP to MES systems using World Batch Forum (WBF) XML schemas standards. Business To Manufacturing Markup Language (B2MML) is an XML implementation of ISA-95 family standards. B2MML comprises of a list of XML schemas developed using the World Wide Web Consortiums XML Schema language (Manufacturing Enterprise Solutions Association (MESA International) 2013).

2.3.9 CMSD

To address interoperability issues between simulation and manufacturing applications the Core Manufacturing Simulation Data (CMSD) product was organized. The CMSD information model provides a data specification for the efficient exchange of manufacturing data in a simulation environment (Core Manufacturing Simulation Data Product Development Group 2010). The data specification can be used to support the integration of simulation software with other manufacturing software applications. The CMSD information model defines the basic

entities in the manufacturing field and the relationship between those entities that are needed to develop manufacturing simulations. Although the information described in this model may be connected with one or more different manufacturing fields such as inventory management, production management or supply chain management, the model is not intended to be a broad definition of the entire manufacturing or simulation domain (Bloomfield et al. 2012). The CMSD information model is given in two modeling languages: (1) the information model defined using the Unified Modeling Language (UML) and (2) the information model defined using the eXtensible Modeling Language (XML) (Leong et al. 2006). The exchange of data between simulations will be allowed through the exchange of XML instance documents that follow the CMSD XML Schemas. The UML diagram and XML schemas are intended to be equivalent representations of the same CMSD model.

Contrary to CMSD, the other contemporary standards (STEP AP214, AutomationML and ISA-95) did not start from the DES area but aim to demonstrate and extend their capabilities for dynamic simulations. They have not been applied to that extend for describing data on the detailed level as required for DES. NIST SDM has been incorporated by CMSD while Riddick and Lee (2008) have introduced enhancements to CMSD specification to support the definition and exchange of layout information corresponding to the manufacturing object types defined by SDX.

2.4 Manufacturing Processes Representation

As identified in the introductory Chapter 1, graphical representations support the visualization of models. This makes it easier for certain users that are accustomed to creating models in a graphical manner, to create, manipulate, and debug models. The graphs are serialized in textual data files and are used to interchange models. Lacy (2006) lists the applications of graphical representations:

- business process representations,
- military process representations,
- representations of general purpose software,

- web-services representations and,
- process interaction discrete event simulation models.

The options available for each representation category listed, are given in Table 2.4, as identified by Lacy (2006). In all approaches, graphical representation involves associating icons with statements and describing control flows as collections of arcs and nodes.

Table 2.4: Graphical Representation Categories (Lacy 2006)

Category	Representations
Business process representations	task networks BPMN (Business Process Modelling Notation), WPDL (Workflow Process Definition Language), PSL (Process Specification Language), process specification graphs, block diagrams
Military process representation	operational templates, BML (Battle Management Language), IDEF family of standards
Representations of general purpose software	STD (State Transitions Diagrams), UML (Unified Modelling Language)
Web-services representations	BPEL4WS (Business Process Execution Language for Web Services), OWL Services, WSDL (Web Service Definition Language)
Process interaction modelling representations	SIMULA activity diagrams, control flow graphs, petri nets, activity cycle diagrams, PND (Process Network diagrams)

In the current research, a similar graphical representation approach is implemented. None of the approaches listed here or in Chapter 1 is adopted as the complexity would impede the simplicity required, as explained later in Section 2.6. The solution proposed is based on the notion that a model is designed by use of a Graphical workflow editor (see Chapter 3). The model is represented thus

as a collection of nodes and arcs connecting the nodes. Each node corresponds to a manufacturing process such as buffers, machines, assembly, workstations, or resources. The arcs represent possible flow of material, or resource assignment to certain stations that compete for these resources. This information is mapped to data structures that capture the logic of the manufacturing processes. In depth description of how the data model and interface is structured are provided in the following chapters.

2.5 Selecting a Data Exchange Format

For the needs of the DREAM project, a fast intuitive way to exchange data is needed. The needs of the study case are mostly specified by the technologies used for the development of the tools produced (ManPy, GUI, KE tool) that need to exchange data. The data exchanged are simple and easily described by native data structures provided by the languages used, namely JavaScript and Python. Thus, a data exchange format that could support data structures expressed that way would have a head start.

Presentation oriented languages are not well suited to cover the specifications, as they are not fit for data serialization. Thus, we can reduce the options available by excluding the presentation oriented languages. Data Exchange Formats that are intended to address needs of specific fields are not considered. The problems that they focus on are not relevant to the main subject of the current study.

It is requested that the format has a well-supported ecosystem of parsers and translators so that it can easily be integrated in different platforms. Referencing within the data structures of the documents is desired but it is not a priority. Schema support is of great importance, as it is crucial for validation of the data transferred, and for the synchronization of the different modules of the platform.

As mentioned in Section 2.1.1, from the options provided in Table 2.1, the most prominent candidates are XML, YAML, and JSON. The rest are either immature and thus their ecosystem of schemas and parsers are not as developed as for the three major competitors, or they lack important features such as schemas or referencing capabilities. Binary load support is not of importance for the

current study. Considering the complexity of YAML, the focus eventually lies on JSON and XML.

Eventually, the format selected has been JSON. The first reason is that JSON is designed to be easy for humans to understand and for computers to parse, introducing also minimal overhead compared to other popular formats, such as the eXtensible Markup Language (XML) (Nurseitov et al. 2009). Within the scope of DES, there is no need to exchange complex data formats such as images or large binary files and thus the use of the not extensible JSON format creates no problems with its limited extensibility when compared to XML. The data structures to be used and described within the data files to be exchanged by the platform modules are no other than records, lists, and trees. In other words, JSON has the potential to provide the same interoperability as XML within the scope of DES.

As the different platform modules are developed in Python and JavaScript, the choice of the JSON data format was preferred to XML due to the native use of the same data structures that JavaScript and Python use. JSON is intuitively compatible with the computer languages used in the DREAM modules, namely JavaScript, the language of which JSON is a subset and Python, which uses the dictionary data type in its core that is a data type similar to JSON objects. Thus, parsing efforts are limited through the use of the JSON format.

JSON is simpler than XML and easier to parse in Python. The parties involved with the development of the DES platform were better accustomed to use with JSON and its smaller grammar that maps directly to Python and JavaScript. On top of that, defining a schema and parsing the exchanged data in the form of JSON will be rendered easier as JSON notation makes no use of tags or attributes. JSON notation is inherently built in Python and JavaScript and therefore no additional software is required.

As migration from XML to JSON and vice versa is straightforward, and industry specific vocabularies can be converted from one to another form, any JSON notation used for the simulation application would be easily translated to XML.

Last but not least, there is no other solution to the problem of DES data exchange that is built on JSON. To our knowledge, there has been no attempt

to set up such a standard using JSON. As we saw in section 2.3, almost all the existing standards that focus on the DES data interchange are based on XML.

2.6 Selecting a Simulation Data Exchange Model

The available standards and interfaces for manufacturing data exchange, in their attempt to include as many different attributes as possible, become too complex requiring thus a large effort to understand and become acquainted with. In order to overcome this constraint, it was decided to structure a new interface that would be easier for a third party developer to understand. Expandability was one of the prerequisites, enabling in that way future inclusion of industrial system complexities.

COTS DES simulation software do support the use of existing standards for DES data representation and exchange, but these standards are not the inherent representation for DES models used by commercial DES platforms. Example implementations show that different standards can be used in combination with commercial tools for manufacturing simulation (Bergmann et al. 2011; Heilala et al. 2011; Leong et al. 2006). On the other hand, commercial DES platforms do not impose familiarity with such model representations as prerequisite from end users in order to develop DES models and exchange data with existing ERP-MES systems. The end user of COTS DES tools can create models graphically and exchange data with external data sources in a simple and intuitive way, without the need to use sophisticated parsers. A similar approach is decided to be adopted in the current research. The directed graph representation (see section 2.4) fits well with such an approach.

Design limitations and constraints that advocate the selection of the proposed approach, will be presented in detail in Chapter 4. First, the architecture of the DES platform used as a pilot for the proposed DES data exchange interface, and the imposed DES information flow constraints are detailed in Chapter 3.

2.7 Conclusions

Aiming to develop a DES platform, a data exchange interface is needed. Initially, after a literature review and based on technologies that are used for the development of the platform, JSON was selected as the data exchange format on which the API would be based. The solution proposed, offers the innovation of developing a data model for exchanging DES related data based on JSON.

JSON has the potential to provide the same interoperability as XML within the scope of DES. JSON data format makes use of the same data structures that JavaScript and Python use. JSON has a well-supported ecosystem of parsers and translators so that it can easily be integrated in different platforms. In addition, schemas are supported by JSON; this virtue is crucial for validation of the DES data, and for the synchronization of the platform.

The structure of the DES API data exchange model has to be decided. The available standards and representations for manufacturing data exchange are to complex and require from developers not familiar with DES nomenclature a large effort to understand and become acquainted with. The concurrent development of different DES platform modules poses limitations to the use of existing data exchange solutions.

It was decided to structure a new interface that would be easier for a third party developer to understand. Expandability was one of the prerequisites, enabling in that way future consideration of manufacturing processes complexities. An interface frame in the form of a JSON data model is established, describing the guidelines that any data exchange between the modules of the platform should follow. The data model aims towards the definition of the format of the input models and the format definition of the results to be displayed to the end-user. The DES models can be described as directed graphs according to this approach.

3

Simulation Platform Requirements

In the third chapter, the Simulation Platform developed within DREAM project is presented. We try to introduce the major objectives of DREAM and the way they guide the research presented within this work.

As stated in the first chapter (chapter 1), the scope of the DREAM project is the development of a simulation platform that based on an expandable and semantic free Simulation Engine (SE). The SE co-operates with other modules such as the Graphical User Interface (GUI) developed in JavaScript, and the Knowledge Extraction (KE) tool. All the modules are structured to be independent but cooperating.

An API has to be developed in order to address that need. The details of that requirement are described as we go through the different modules of the platform and the related application specifications.

3.1 Simulation Platform Objectives

As stated in the introductory chapter (section 1.4), the current research is conducted within the scope of the DREAM project. DREAM intends to build a simulation application platform enabling the ease-of-use application of scenario-based DES in day-to-day business at small manufacturing companies and large industrial enterprises at reasonable costs (Heavey et al. 2014).

The main requirements are given in the following list:

- The simulation platform must be able to run DES scenarios.
- Certain models may need to have knowledge extracted from various sources. It is of high importance that the platform maintains a level of flexibility that allows integration of different sorts of information sources.
- End users need to interact with a simple user interface that enables them to review results, and enter input data for the simulation models.

To fit objectives, a platform with a simulation main server as its core component is used. The high level architecture of the simulation platform is depicted in Figure 3.1. It is running Python and uses three sub-components, all developed for the needs of the platform. The code of the modules and documentation exists online in the project repository on Github (`http://github.com/nexedi/dream`).

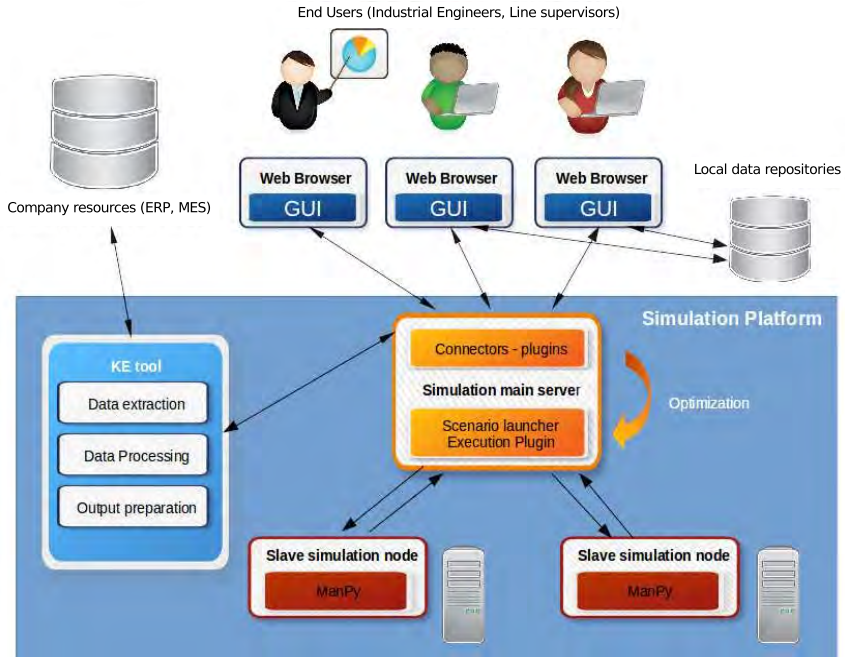


Figure 3.1: Simulation Platform Design - High level architecture of the platform and the simulation server

The platform is cloud enabled and is designed to leverage of modern HTML5 (Hickson et al. 2014) and JavaScript for the user interface. The simulation framework is a Python library that uses the SimPy DES library (Team SimPy 2015).

The communication with the simulation server is based on HTTP (Fielding et al. 1999) for the transport protocol. The main components of the platforms are described in the following section.

3.2 Platform components

3.2.1 Simulation Engine - ManPy

Evaluating the available open-source simulation tools, SimPy (`http://simpy.sourceforge.net/`) was selected as the most suitable for the needs of the platform Dagkakis et al. 2013. On top of SimPy, a layer of manufacturing objects was developed. That layer is referred to as ManPy.

ManPy stands for Manufacturing in Python and is a layer of well-defined manufacturing objects that is built in the Python programming language (Python Software Foundation 2010). ManPy utilises the SimPy DES framework that provides the infrastructure of implementing processes that progress in parallel in a process oriented DES world view. The main scope of ManPy is the provision of prefabricated simulation components, which a modeller can use to build a simulation model, in the same fashion as most COTS DES software tools work. Moreover, given the OS nature of the platform, advanced users have the capability of populating the repository and exchanging objects.

3.2.2 Knowledge extraction module - KE tool

KE tool stands for Knowledge Extraction tool and is a tool built to link production data stored in different organizations' IT-systems with the simulation software. Three are the main stages that have been identified for the implementation and use of that tool (Figure 3.2):

- First, the data extraction which is the procedure that has to be done in order to obtain raw data from company resources (ERP or MES systems).
- After the extraction, calculations may be needed to transform the data samples into useful form. Basic statistical analysis and data mining methods,

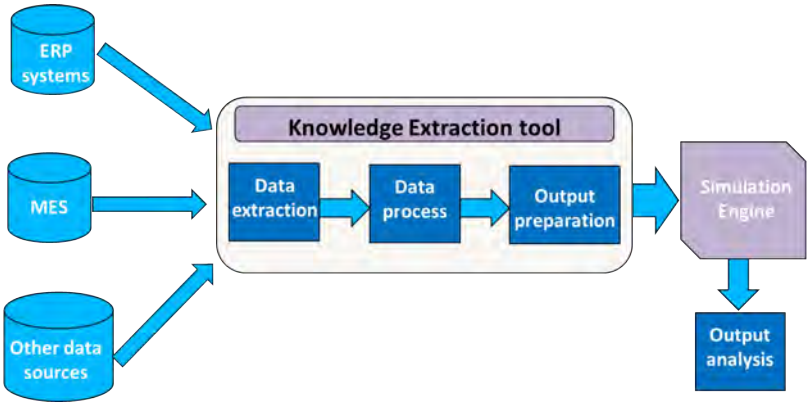


Figure 3.2: KE tool - Architecture of the KE tool

wherever they apply, are performed. All these calculations constitute the second stage which is named data processing.

- Another stage is the output preparation. The data are transformed into a format that can be used as an input in the simulation platform.
- The last stage of the data preparation for the simulation core is output analysis. Statistical analysis of output data takes place.

RPy2, which is an interface between Python and R scripting language along with other Python libraries were chosen to be the core for the development of the tool (Gautier 2008). The basic requirements that were followed in the development process is that the KE tool should be modular, extensible, open-source and flexible. Considering the above, objects covering the core components were developed (Data Extraction; Data processing; Output preparation; Output analysis) of the tool. Every object has its own attributes encapsulated in its code and outputs its results when is called to be executed.

3.2.3 Graphical User Interface

The platform offers the modellers the capability to create and modify simulation models in a graphical way similar to COTS software. In addition, it is requested that the Graphical User Interface (GUI) is accessible via the Web, removing the restriction to run the platform from a single workstation. On the contrary, the

simulation platform should be accessible from everywhere, as long as internet access is granted.

Figure 3.3: Graphical User interface - Example of the DREAM graph editor

The capability to run the GUI on different platforms is also to be provided (Figure 3.3). Thus, the DREAM GUI design is based on the JavaScript programming language. Summarizing the requirements, DREAM GUI has been developed with two key ideas in mind:

- the first goal is to help the user to develop a simulation model visually (Figure 3.3),
- the simulation experiments results should be returned to the end user in a straightforward and understandable manner that should help him comprehend at a glance the output of the model by presenting the results in formats such as bar charts, plots, Gantt diagrams or spreadsheets (see Figure 3.4).

3.3 Architecture and Information Flow

All three modules of the platform must be interfaced in a flexible, extensible, and neutral way. The application interface to be implemented should guaranty the modularity and the extensibility of the platform.

Figure 3.4: Graphical User interface results - Example of a chart for results analysis

In a typical instance of the platform, the three modules communicate as depicted in Figure 3.5. The platform should be instance-independent and different features should be supported regardless if they are eventually invoked in any use-case specific instance. There are two use-cases identified and both can be described by Figure 3.5:

- In the first one, the users have in depth understanding of the platform and can decide on the features they need. The end users should ideally select the features and the modules to be used in the instance they implement. That can be achieved either by modifying a file that contains relevant info, or graphically. Enabling the required features, the end users can then interact with the platform in the desired way. They can design or modify a given model, collect data from an external data source, or use graphical means to input data manually, request optimization or single simulation runs, and eventually review, evaluate, and compare simulation results.

- In the case of tailored solutions, where the end user is provided a model and an instance of the platform, the different features of the platform (widgets of the GUI, translators for the data collected, object classes for the SE) should be synchronized according to the needs of the end users. These needs are identified by the expert modellers and the features to be used are chosen by them.

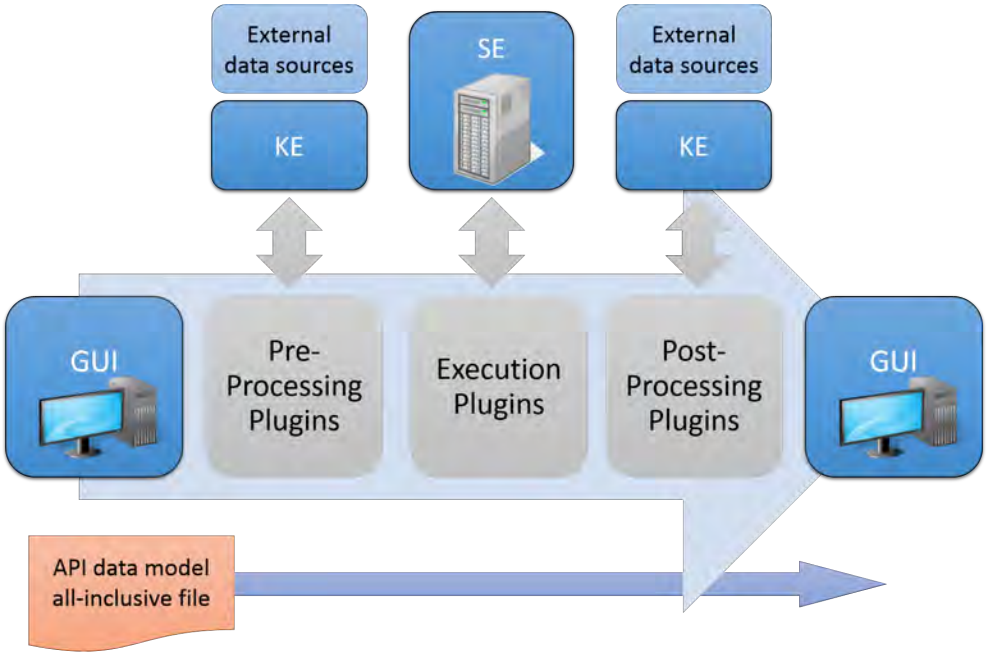


Figure 3.5: Information flow - Interfacing of the three modules of the platform

For both cases, information is collected initially from the GUI. The end user selects the features, he will use. The user then describes the models in terms of a directed graph. Additional information is gathered from the GUI and external data sources. The KE tool performs data analysis and different plugins translate the information from the various repositories into the format defined by the data schema. Then the SE is fed with the information collected as prepared by the translators. The raw results are introduced to different output preparation plugins and eventually returned to the GUI to be reviewed by the user.

For the above mentioned reasons, it is considered ideal that an inclusive file should contain instance related information, the model, and collected data from external data sources. All that information should be gathered sequentially and then collated in a single file. The procedure can be described, defined, or guided by a data model.

All the different plugins that do process in any way the input or output data are invoked serially as defined by the user/modeller within the data model. Even the SE or the optimizer are invoked as plugins. The different plugins can be categorized as:

1. preprocessing plugins,
2. execution plugins, and
3. post-processing plugins.

3.3.1 Preprocessing Plugins

The platform will execute one after the other each of the preprocessing plugins before executing the scenario by the SE. Typically, the preprocessing plugins can be used to prepare the data before running the scenarios. They can interface with the KE tool methods. Eventually, the KE tool can be described in terms of plugins where each plugin can perform data-extraction, -processing, -analysis, and output preparation.

The first family of pre-processing plugins usually connects to some external application to read data. Writing pre-processing plugins is made easy by reusing knowledge extraction objects and methods to i.e. connect to a database or run distribution fitting tests.

For example, preprocessing plugins should connect to a data base and populate the work in progress levels of each buffer object in the model before running simulation by reading this information from an MES database. In addition, the plugins should preform calculations and analysis of the data collected. For example, it may be requested that the distributions for the processing time of the stations are calculated by running a Kolmogorov-Smirnov test (Smirnov 1944) on the historical data of processing times that have been recorded in the MES.

The second family of preprocessing plugins will pre-format the data in a format that is suitable for the SE according to a schema that should constitute part of the data model. For example, the integrated platform should feature some simple spreadsheets to define the work in progress, bill of materials, and machines and/or employees shifts.

3.3.2 Execution Plugins

An execution plugin is a script (class in terms of object oriented programming) that will execute the simulation scenario using ManPy DES library. In the default case, it will simply execute the scenario and return the result, but this plugin can

be replaced by another one that would execute variations of the scenarios that can be later compared. This is also the place to perform optimisation with simulation.

3.3.3 Post-processing Plugins

Post processing plugins are invoked after the scenario has been executed on the SE. These plugins calculate various key performance indicators such as the number of entities produced per day, the utilisation ratios of available resources or the work in progress levels in different buffer objects.

Data preparation for visualization is also performed by post-processing plugins. For example in a job shop use-case, the SE will output the route to be followed by each of the entities (e.g. order components) to be produced, a post processing plugin can reformat the data to match the format expected by a GANTT visualisation widget or any other chart widget (GUI).

Post-processing plugins can be used to interface with external data stores in order to store simulation results for later use. For example, they can store the result of simulations in a database to build a set of simulation data that could be used to implement Just In Time Information Retrieval (JITIR) (Rhodes 2000).

3.4 Conclusions

The issue of data sharing between manufacturing and simulation applications, internal or external to the DES platform has to be addressed. Diverse modules have to be interfaced in a neutral and semantic-free way. DES data is to be collected (exported) from (to) different modules of the platform as well as from (to) external manufacturing applications. The interface should guarantee the interoperability between the different modules of the platform.

In order to be consistent with used technologies, it is required that they are open-source. Given that the API should guarantee interoperability for modules and applications unrelated to DES terminology, it is desirable that the API is easy to learn by parties not familiar with the nomenclature of DES, and easy to adopt to the specific needs of the modelling.

One major concern for the development of the API should be the expandability and reconfigurability, so that the platform can be modified and the different modules synchronized only by the use of the API.

Finally, it is required that the API and the data model are developed to provide a data representation for DES models that resembles the COTS product's CAD (Computer Aided Design) representation. As depicted in Figure 3.3, the GUI graph editor module describes DES model topologies as directed graphs, in other words, as collections of nodes and edges. Similar approaches are common among COTS simulation tools and modellers are familiar with that way of designing models.

4

DES Application Programming Interface data model

A JSON data exchange model was structured aiming to describe a simulation model, to configure a DES platform, and exchange data between different internal modules of the DREAM platform or external modules. The data representation chosen for the description of the simulation models is based on the directed graph definition (Chartrand 2006). A JSON schema was structured aiming to describe a simulation model as a directed graph. The schema is build according to the definition of the draft JSON schema (Galiegue and Court 2013).

Initially, the constraints and limitations that decided the selection of the development process of the JSON data exchange model are listed. The scope of the data model presented in this work is to provide a meaningful data structure description for the modules within the DREAM platform for different use case scenarios. The first sections of the chapter describe the structure of the data model used to describe and exchange DES models and DES related data. The second half of the chapter focuses on the structures used to synchronize and configure the DES platform.

4.1 Development considerations

The concurrent development of a core simulation tool (ManPy or Manufacturing in Python) and DES related modules, as for example a process workflow editor

(Graphical User Interface), poses limitations to the use of existing data exchange formats or to the development of any new data exchange formats. As explained in section 1.4, and Chapter 3, the interface provided should be programmable, extendable, and easily adaptable to the needs of various use cases.

Contrary to the established practice which would require the definition of a work-frame that would act as a guide for the choice of a data exchange file format and the adoption of an existing or the development of any new data exchange interface, the data exchange file format was first decided. As discussed in section 2.5, JSON was the obvious choice for the data exchange file format. Due to the involvement of a third party developer, a more agile approach was preferred, and a new API for DES simulation data exchange was developed. That allowed the third party to get quickly accustomed to the notions and the needs of manufacturing processes simulation environment. On top of that, an interface frame in the form of a JSON data model was established, describing the guidelines that any data exchange between the modules of the platform should follow.

Through the development of a Graphical User Interface workflow editor (see Figure 3.3), the idea of simple generic data model describing a simple manufacturing process work-flow was conceived. In short,

1. Simple rules for the creation of the JSON files to be exchanged were adopted as ManPy and the GUI were being developed and expanded.
2. Those files had to be enriched in the process with new model objects and the data model and interface started to become more complex as the data structures described within the models were populated.
3. The rules were also rendered more complicated and hard to describe. In that respect, the need to adopt a general JSON schema that would inclusively describe the data exchange for every possible pilot case emerged.

The development and use of such a data model for the API of the platform should make easier the development of the simulation platform itself. The establishment of a well-structured data model aimed towards:

- the definition of the format of the input models to be simulated as described by the user in the workflow editor,

- the format definition of the simulation results to be displayed by a graphical user interface,
- capability to incorporate the input requirements of the objects available in the repository of the simulation environment in terms of required inputs from the work-flow Editor.

The last requirement, regarding the definition of the input requirements of each entity or process described in a simulation model, can be addressed by an embedded JSON schema. Such approach can render the synchronization of the platform easier for potential developers. In this way, any new object in the repository of the DES platform can be automatically described in terms of required input parameters within the schema, and subsequently any GUI instance making use of the specified object can be updated according to the input needs of the selected modelling object.

Following this schema, the interoperability between the existing modules of the platform or any future module is guaranteed. For the purpose of data exchange between an external data source (e.g. an ERP system) and the simulation application, an intermediate step of model definition is required through the use of a work-flow editor. That model can be described using the JSON data structures, complying with the JSON schema guidelines provided. It should be noted that objects describing the logic of the corresponding processes modelled should be developed beforehand.

4.2 Data exchange model layout

The layout of the data model in relation to the platform structure is presented in Figure 4.1. The information flow presented in Figure 3.5 can be expanded to Figure 4.1 as explained in the following sections. Four basic structures of the data model (*general*, *graph*, *input*, and *results*) can cover the data exchange interface needs between a simulation platform and any external modules. General attributes concerning a simulation experiment can be described in the *general* properties. The topology of a DES model in terms of directed graph nomenclature constitute the *graph*. Extra information about moving DES entities (order

parts, components, batches, etc.) and relationships between them, as well as initial conditions are described by the general term *input*; these entities cannot be incorporated in a model graph. Eventually the simulations results are returned by the simulator as a general *result* structure. This information is sufficient for the description of DES model, of the initial conditions, and eventually the results calculated.

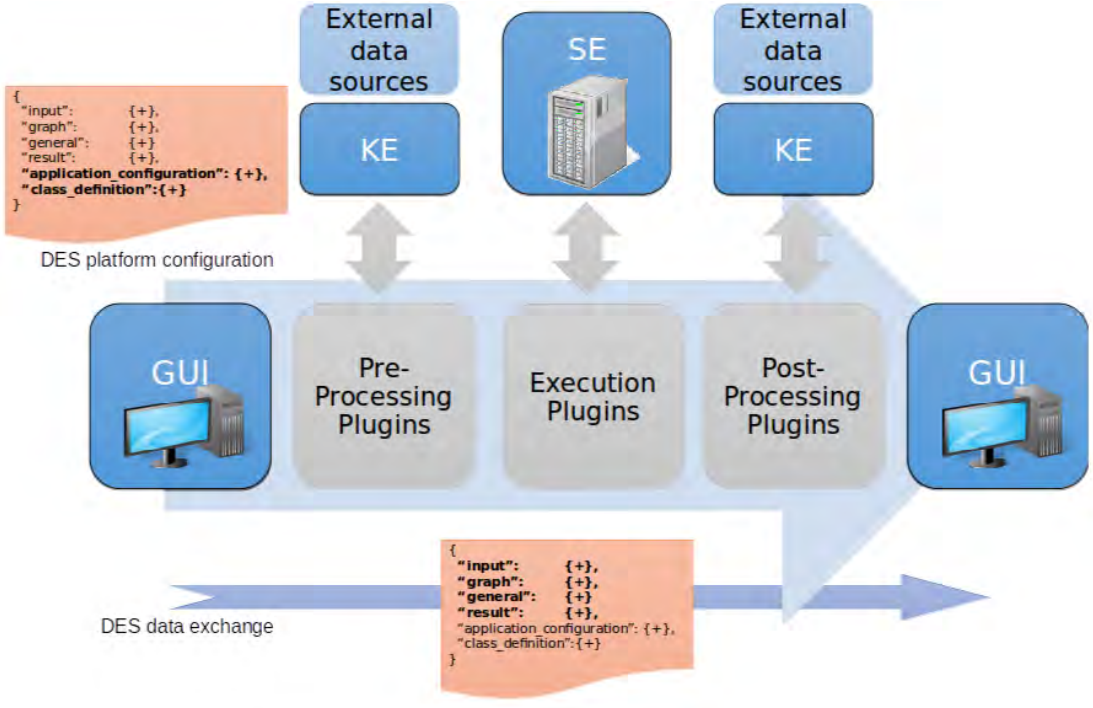


Figure 4.1: Information flow - Interfacing of the three modules of the platform

Additionally, definitions and configuration data that have to do with the DES platform synchronization are included in the data model. *Class_definition* and *application_configuration* dictate the way the platform is synchronized, DES related information is collected, parsed, and pre- or post-processed. The instance of the GUI is created following the *application_configuration* and *class_definition* instructions. The *graph* is generated in the GUI workflow editor according to the rules defines in *class_definition*. The *class_definition* structure is used as an embedded schema for the creation of valid DES data.

Another way to depict the data model is by means of class diagrams. Using

such an approach, the data model has the general form depicted in Figure 4.2. It is easily noticeable from the layout of the schema that there is an explicit differentiation between the different applications and uses of the data model.

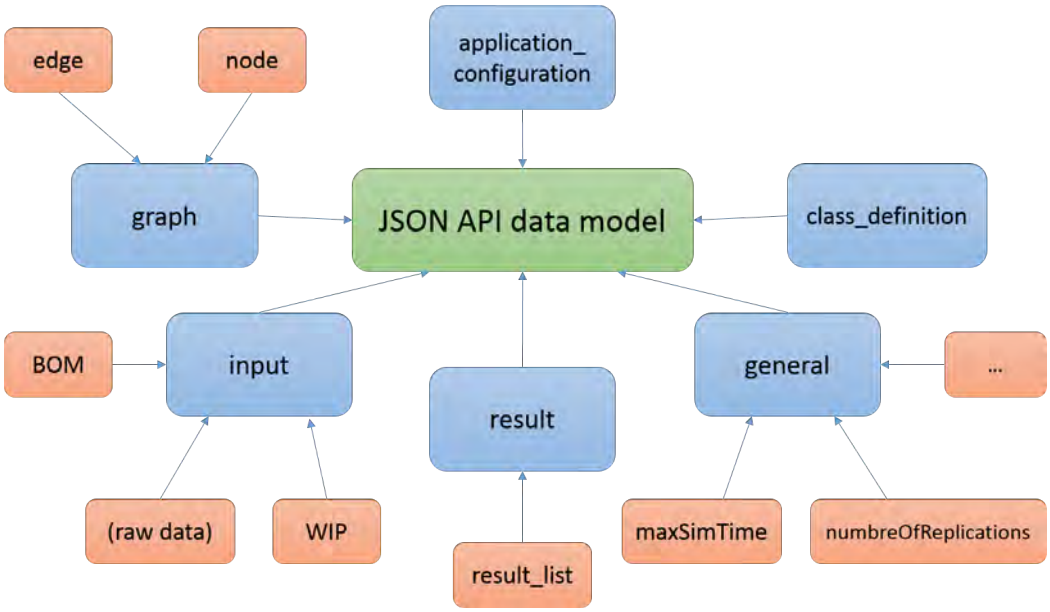


Figure 4.2: JSON API structure - Structure of the JSON API data model

A JSON file sent from the external module containing the simulation model description consists of at least four structures (see Figures 4.1 and 4.2). A detailed version of Figure 4.2 is provided in Figure 4.16.

4.3 DES data exchange model

The basic parts of the data model according to which the file is structured are *graph*, *input*, *general*, and *result* (Figure 4.3). *General* refers to attributes that a simulation experiment needs for each run. This may refer to the number of replications, the length of each replication etc.

Graph First, the JSON file should be in position to describe the topology of the DES model in terms of directed graph nomenclature. Thus, in each data exchange file there is a collection of nodes and edges under the *graph* (Figure

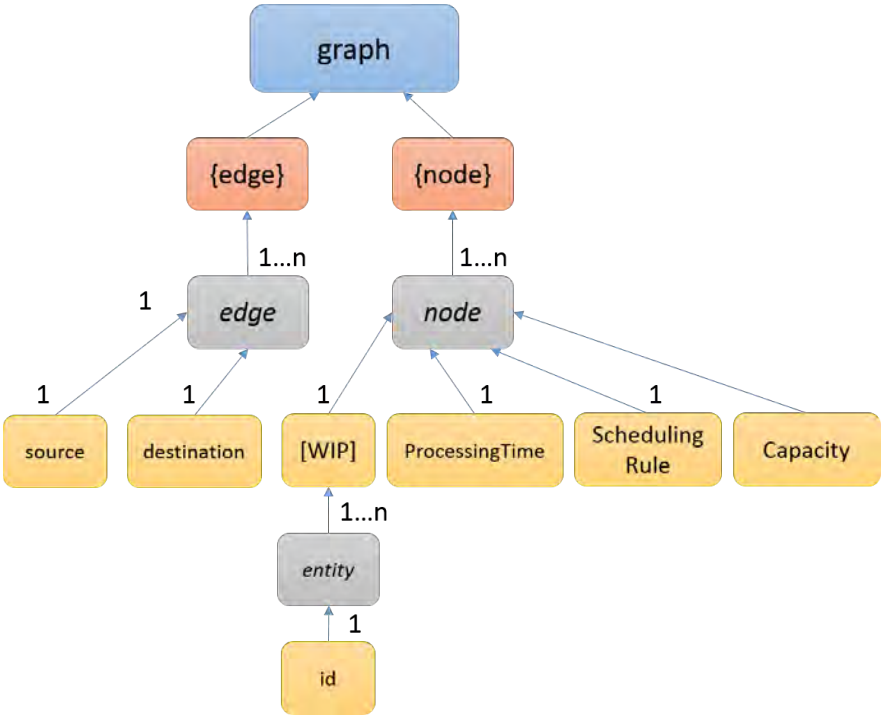


Figure 4.3: Graph layout - Simple layout of the graph representation

4.3). *Graph* is the JSON object that describes the topology of the DES model. Each node represents an object that is permanent/fixed for a DES model (e.g. a buffer, a server an assembly point etc.), and not a moving entity. These are described as JSON objects and keep all their attributes as encapsulated JSON objects. Every edge describes the connection between two nodes one of which is the starting and the other is the ending point of the connection.

Every different node has a selection of various properties. For example, a machine can have a a *processing time*, while a buffer has a *scheduling rule* property according to which it sorts the entities it holds. In Figure 4.3, the JSON object *node* (red colour) holds a set of nodes ($1 \dots n$, grey colour), everyone of which can be a machine, a buffer etc.. Each of these nodes, can have one or more properties.

Input The object *input* holds input information that cannot be classified under the *graph*. This object can act as a placeholder for raw information collected by external data sources, before they are processed, and given a structured format

so that they can be parsed by the SE.

Additionally, the *input* object can store information on entities that cannot be described as nodes and edges. Entities that move within the modelled DES Moving Entities (MEs) system may be structurally related to other entities, i.e. comprise a specific order (components that have the same parent order). In Figure 4.4, we see that an *input* object called Bill Of Materials (*BOM*) is used to describe these data structures that cannot be uniquely identified within the system as fixed objects/nodes. These entities cannot be described within the *graph* object. Entities that reside in the simulation model when it is initiated, can be defined within *input* under the object named *WIP* (Work In Progress). The *WIP* and *BOM* objects are predefined structured properties of the *input* object, the same way that nodes and edges are properties of the *node* and *edge* objects (*graph*). The difference can be simply described as:

- every node and edge are characterized by a unique key (node and edge id), eg nodeID1, nodeID2, nodeID143,
- *BOM* and *WIP* objects are unique for the *input* object. On the other hand though, *WIP* can hold a collection of entities, each one of them being unique (having a unique id).

Result After any simulation run, relevant results information should be retrieved by the modeller in order to be evaluated. The structure depicted in Figure 4.5 defines the structure of the results returned to a module (e.g. GUI or KE tool). The data sent from the simulator core containing the simulation output populates the encapsulated JSON object *result*. The *results_list* object contains a list of solutions (one or more) with information on any entity. MEs, such as jobs contain routing information as calculated by the simulator, while servers as stations or buffers provide information about their working or blocking times.

The four major parts of the API data model just presented, can be used separately from the others introduced in section 4.4 on page 67. The simulation engine of the platform (ManPy) can interface with external DES modules by the

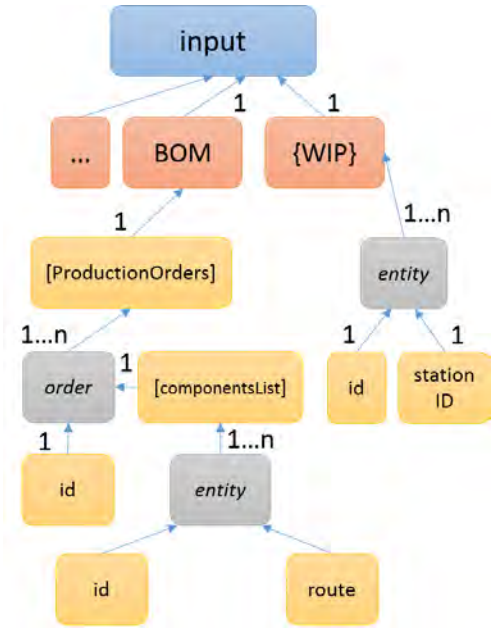


Figure 4.4: *input* layout - Structure of *input*.

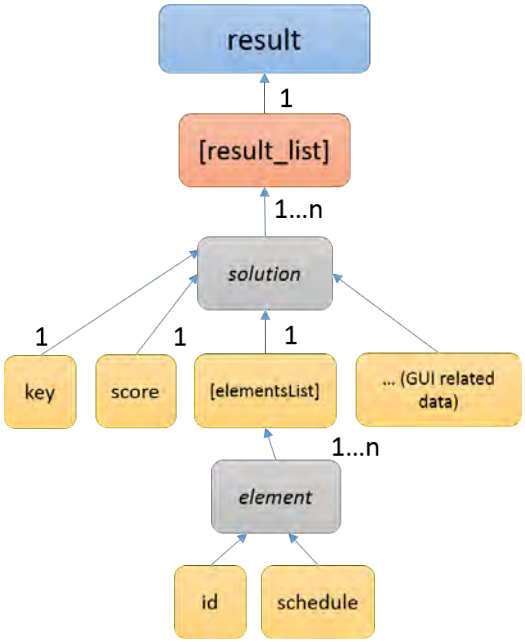


Figure 4.5: *result* layout - Structure of *result*.

use of these four JSON objects (*general*, *graph*, *input*, and *results*). Appropriate parsers can be created for installations with other DES simulation engines.

The DES API objects that are used for the configuration and synchronization of the DES platform (*application_configuration*, *class_definition*, in Figure 4.3), are introduced in detail in section 4.4 on page 67.

4.3.1 Directed Graph model representation

A simulation model is structured following a graphical representation schema in order to describe a simulation model as a directed graph. The schema has the form of Figure 4.3. The basic units for the representation of a model are nodes and edges.

Nodes and edges are common conventions used in workflow editors to describe topologies where information is exchanged between nodes as defined by connecting edges. The notion of nodes and edges is also implemented in Graph Databases where graph structures with nodes, edges and other properties represent and store

data. Nodes are used to model stationary parts of manufacturing processes and edges can describe the flow of MEs among the stationary parts.

4.3.1.1 *Node*

Under the key *node* in the data model, a dictionary object is structured. This JSON dictionary contains a collection of key-item pairs. The key of every item corresponds to a unique *ID*. Every item can be a valid JSON object of its own. The properties of the item are contained within that object. Each of these key-item pair represents a node.

Every one of these nodes corresponds to an object that is permanent/fixed for a DES model (e.g. a buffer, a server an assembly point etc). These are JSON objects and store all their attributes as encapsulated JSON objects. Every node should necessarily have two attributes: first, an *ID* that would be unique for this specific node and it is the key of the JSON object that describes the DES object. Second, a *_class* attribute, that corresponds to the class of the simulation object that the node describes. For example the definition of a Queue simulation object from the DREAM repository with capacity of one can be as simple as the one shown in Listing 4.1.

Listing 4.1: Node example

```
"node": {
  "Q1": {
    "id": "Q1",
    "_class": "Dream.Queue",
    "capacity": 1
  }
}
```

In addition to these two basic attributes, the JSON counterparts of the DES objects have extra properties. For example a Machine, may have a *processingTime* attribute that is also a JSON object that describes the way the processing time of the machine is defined. An example is given in Listing 4.2. Different distributions may require different attributes (e.g. standard deviation) and more complex distributions, such as empirical ones, may be defined.

Listing 4.2: Processing time of Machine object

```
"node":{
  "M1": {
    "_class": "Machine",
    "name": "Station 1",
    "processingTime": {
      "Fixed": {
        "mean": 1.5
      }
    }
  },
  "M2": {
    "_class": "Machine",
    "name": "Station 2",
    "processingTime": {
      "Normal": {
        "stdev": 0.3,
        "mean": 1.5
      }
    }
  }
}
```

A node may also have a *WIP* attribute to define the starting Work In Progress at the start of simulation. Nevertheless, jobs or similar movable entities may have specific interconnections between them, for example comprise a specific order. For this reason, Bills Of Material (*BOM*) echelon is introduced under *input*, as we will see in section 4.3.3.

In terms of GUI, a simple model presented in Figure 4.6 can be described easily following the graph representation data model. The corresponding *node* object of the JSON representation of the model is given in Listing 4.3.

***Node* object schema:** It is possible to document the structure of the JSON *node* object using JSON schema notation (Crockford 2006). Applying this approach, it is easy to track the key properties of numerous different objects, as well as any changes in the format that the data structuring should comply with. Listing 4.4 defines how the *node* object should be formatted.

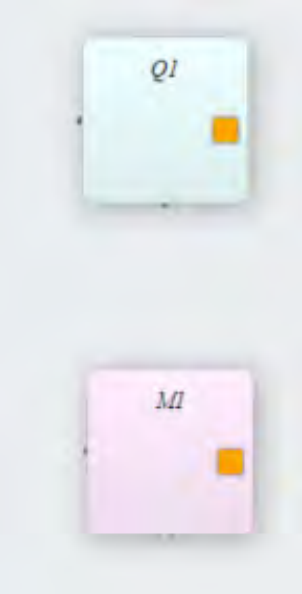


Figure 4.6: Simple nodes graph - A buffer and Machine as nodes

```
"node": {
  "M1": {
    "_class": "Machine",
    "name": "Machine1",
    "processingTime": {"Fixed": {"mean": 1}},
    "id": "M1"
  },
  "Q1": {
    "_class": "Dream.Queue",
    "name": "Q1",
    "capacity": 1,
    "schedulingRule": "FIFO",
    "id": "Q1"
  }
}
"edge":{}
```

Listing 4.3: Simple nodes example corresponding JSON representation (Fig. 4.6)

Listing 4.4: *node* object JSON schema

```
"node": {
  "type": "object",
  "patternProperties": {
    "[A-Za-z0-9]":{"oneOf": [
      {"$ref": "#/class_definition/node"},
      {"$ref": "#/class_definition/Queue"}
      {"$ref": "#/class_definition/Machine"}
    ]}
  }
}
```

The *node* object is a dictionary of JSON type *object*. The different nodes or in other words, the items defined within the *node* object should comply with the definition provided under the *patternProperties* property of the *node* schema. According to that, their keys can have any alphanumeric value ($[A-Za-z0-9]$), for example "Q1" or "M1". The corresponding items' value of every key-item pair should follow the definition provided by the object at position

"#/class_definition/node" or elsewhere (e.g. *"#/class_definition/Queue"*).

JSON schema supports referencing and the key *"$ref"* can be used for that purpose. In such a case, the key *"$ref"* points to the definition of a node or any other object that can be used in the simulation. The reserved key *"oneOf"* requires the corresponding object must be valid against only one of the options provided in the list. Examples of different object definitions will be presented in more detail in subsection 4.4.1.

4.3.1.2 *Edge*

As shown in Figure 3.3 from chapter 3, different nodes are connected graphically by edges. A simpler example can be seen in Figure 4.7. The buffer "Q1" is connected to the station "M1". Different entities stored in "Q1" will be delivered to "M1". The corresponding edge presented in JSON notation can be found in Listing 4.5.

Every edge describes the connection between two nodes. Thus, considering a Queue with *id* "Q1" that is the place where parts are buffered before getting processed by Machine with id "M1", the edge will be defined as in Listing 4.5.

"con0" is the unique *ID* of the edge. Of course, "Q1" and "M1" should correspond to nodes of the same model configuration. More complex routing complexities (e.g. a buffer may forward a job to different stations depending on the job type) can be described within the nodes and the simulation class they correspond to.

Figure 4.7: Simple nodes graph - A buffer and Machine connected by an edge

```
"node": {+}
"edge":{
  "con0": {
    "source": "Q1",
    "destination": "M1"
  }
}
```

Listing 4.5: Simple edge example corresponding JSON representation (Fig. 4.7)

***Edge* object schema:** Similarly to what is presented in subsection 4.3.1.1, there can be a JSON schema for the *edge* object. Listing 4.6 provides the definition of the *edge* object. As discussed in 4.3.1.1, the key of the key-item pair for an edge can be of any alphanumeric value (e.g. "con0"). The corresponding item value must comply with the definition of an edge given at the position *"#/definition/_edge"* (see subsection 4.4.1).

Listing 4.6: *edge* object JSON schema

```
"edge": {
  "type": "object",
  "patternProperties": {
    "[A-Za-z0-9]":{"$ref": "#/class_definition/edge"}
  }
}
```

4.3.2 *General*

The "general" object contains general DES input information required to initiate a simulation run. These can be "timeUnit" for the time units used by the simulator, "confidenceLevel" for the confidence level of the statistical analysis, and various others depending on the type of analysis requested. In Listing 4.7, an example of a "general" object is provided. Different properties with information important for the analysis to be performed are listed. Extra information on the different properties of the "general" object can be found in Appendix B.

Listing 4.7: *General* example

```
  "general": {
    "numberOfReplications": 1,
    "processTimeout": 20,
    "seed": "",
    "timeUnit": "hour",
    "maxSimTime": 5000,
    "currentDate": "2014/07/14",
    "throughputTarget": 10,
    "confidenceLevel": 0.95
  }
```

4.3.3 *Input*

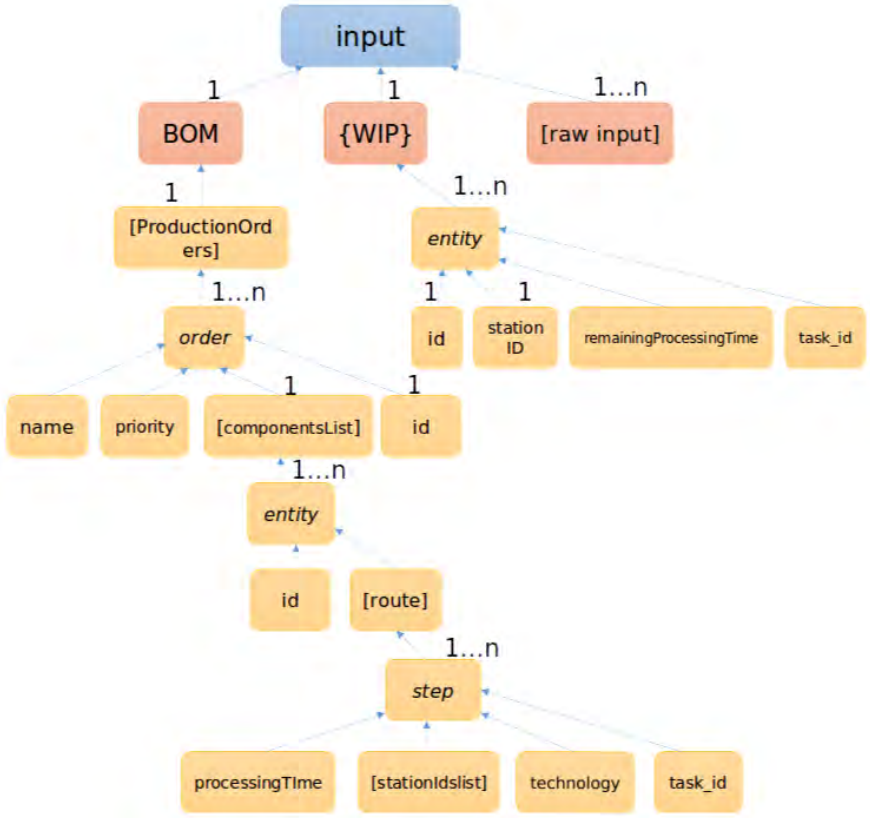


Figure 4.8: *input* layout - closer look to *input* layout

Unprocessed data can be stored by the data model within *input* object. In Figure 4.8, the layout of *input* is presented in more detail than in Figure 4.4. It can contain raw data in different formats that can be described in JSON; Work In Progress (WIP) information, production orders data of a job-shop, or any other info that cannot be directly integrated into *graph*.

For example in Listing 4.8, an array named "wip_spreadsheet" can store information concerning the Work In Progress within a job-shop. The WIP info can be introduced to the platform via the GUI interface (see section 4.4.3). Different kind of information can be valuable for the definition of the work in progress; the station were the movable entity currently resides, the time that the entity entered the station, the currently performed task. That information can be later

processed by a plugin and eventually be attached either to *graph* nodes as WIP information, or to a "WIP" dictionary of all the system moving entities within *input* object.

In a similar way, movable entities of a job-shop DES model (i.e. order components comprising a wider production order) can have interconnections between them. For example, they can have the same parent order; or an order component cannot be processed before an other component has concluded a certain task. The info can be provided in a graphical way through the GUI (ie spreadsheet), or extracted from an external relational DB. In Listing 4.9, an example of an input object containing a list of different production orders is shown. Each order contains a number of components, and every component has to fulfil a number of steps in a certain order before the order is completed.

A "Bill Of Materials" or "BOM" is used to describe the data structures that is not linked to the topology information contained in a graph (*graph*).

Listing 4.8: *input* raw data arrays

```
{
  "input": {
    "wip_spreadsheet": [[
        "Part ID",
        "Sequence",
        "task ID",
        "Station ID",
        "Personnel ID",
        "Start time"
      ],[
        "PE001068A",
        "11",
        "ID-00019",
        "INJM2",
        "W3 - CV",
        "13:30"
    ]],
    "production_orders_spreadsheet": [+],
    "workplan_spreadsheet": [+],
  },...
}
```

In order to understand and demonstrate how the "BOM" can be structured, a system that processes orders; every order consists of a set of different parts, which are processed independently. Constraints related to the planning of every order dictate the sequence in which the components are processed, as well as various other parameters.

Listing 4.9: *input* BOM data structure

```
{
  "input": {
    "BOM": {
      "productionOrders": [{
          "name": "Gelenkstangen 50/0\u00b0/180\u00b0",
          "id": "W134188",
          "priority": "0",
          "_class": "Dream.Order",
          "componentsList": [{
              "_class": "Dream.Mould",
              "id": "PE001068A",
              "route": [{
                  "technology": "ASSM",
                  "task_id": "ID-00018",
                  "sequence": "10",
                  "stationIdsList": ["ASSM1"],
                  "requiredParts": ["all"],
                  "processingTime": {"Fixed": {"mean": 3.0}},
                  "operationType": {
                    "Load": 1,
                    "Processing": 0
                  },
                  "operator": {"load": ["W3"]},
              },...]
          },...]
      },...]
    }
  }
}
```

By examining Figure 4.8, we notice that the BOM contains a "production Orders" dictionary which defines a list of orders that have to be produced by the modelled system. Every order has a "name" and an "id". The "id" must

be unique for identification. Similarly to *graph* nodes, every order has a "_class" attribute. Extra properties can be introduced for DES purposes; "priority", and "orderDate" can be used for scheduling.

Every order consists of several components which are defined by the "componentsList" property. Every component has unique "id", as well as "_class". The components as moving entities should follow a predefined route within the model. Every step of the route corresponds to a "technology" as an action that must performed on the component. That action should take place at specified stations-nodes ("stationIdList"). Every step of the route is uniquely defined by a "task_id" and a sequence that are used to identify where exactly lies the component within its route. Every step may include different operations; loading, set-up, normal processing. For each of these, an "operationType", an operation time ("processingTime", "setupTime"), an "operationType" (manual-1, automatic-0) may as well be defined.

In simpler cases, the "BOM" may contain different lists, as for example a "jobList" for a job-shop, simpler than that shown in Listing 4.9. In such a scenario, jobs can have their own routes, but are not interconnected under a parent order. An embedded schema for the description of the "BOM", or for any relation between orders, entities, components, and nodes of the *graph* could also be introduced to the API data model. In the current work, such an approach is not considered.

4.3.4 *Result*

At the end of any simulation run, the DES results are retrieved by the modeller in order to be evaluated. As seen already in Figure 4.5, the *result* object contains a list of results for the different simulation runs - "result_list"; each one of the results corresponds to a different solution.

Every solution is characterized by a unique "key" which is used for identification. A "score" is also available for the evaluation of the result and comparison with other solutions. The "score" rating is calculated by the optimizer, in case optimization is performed by the SE. Additionally, every solution contains a list of all the elements (stationary-nodes, MEs) of the model in "elementsList". MEs,

such as jobs, contain routing information as calculated by the SE, while servers as stations or buffers provide information about their working or blocking times. In Listing 4.10, we can see an example of a single solution "result_list".

Listing 4.10: *result* example

```
{
   "result": {
     "result_list": [
       {
         "_class": "Dream.Simulation",
         "key": "solution1",
         "score": 0,
         "elementList": [
           {
             "_class": "Dream.Operator",
             "family": "Operator",
             "id": "Worker1",
             "results: {
               "working_ratio": [50],
               "waiting_ratio": [50],
               "schedule": [
                 {
                   "entranceTime": 4,
                   "exitTime": 14,
                   "stationId: "Station1"
                 },...
               ]
             }
           },
           {
             "_class": "Dream.OrderComponent",
             "family": "Job",
             "id": "Component0",
             "results: {
               "completionTime": 30,
               "schedule": [
                 {
                   "entranceTime": 0,
                   "exitTime": 4,
                   "stationId": "Station2",
                   "task_id": "task01"
                 },...
               ]
             }
           },
           {
             "_class": "Dream.JobShopMachine",
             "family": "Server",
             "id": "Station1",
             "results: {
               "working_ratio": [+],
               "failure_ratio": [+], ...
             }
           }
         ]
       }
     ]
   }
 }
```

Every element of the "element_list" has an "id" property pointing to an item of the *graph* or the "BOM" dictionary. A "_class" property corresponding to the SE objects repository is also available. For the classification of the elements,

a “family” property is used. The actual results are contained within “results”. Different ratios are calculated and returned by the SE in the form of arrays (e.g. “working_ration”).

For MEs or resources (eg operators) that are shared between different stations, a “schedule” property contains routing information. For each step, an “entranceTime” and an “exitTime” is recorded, together with the “stationId”, and a “task_id” if available. The “results” dictionary of different elements can be expanded and populated with information related to the specifications of every element.

4.4 DES platform configuration and synchronization

Definitions and specifications concerning the DES platform related information for synchronization purposes are as well included in the data model. Notice in Figure 4.2 that the corresponding DES API model objects are *class_definition* and *application_configuration*. Information maintained by *class_definition* can be classified as referring to global modelling attributes of the simulator (e.g. a list of all simulation objects available in this instance of the SE). It is unrelated to specific simulation models but can be seen as starting point for any DES model initiation as it contains SE related information about available object classes that can be used (e.g. an extension of a basic machine class implementing new functionalities), or the data structure that any initial condition must have, etc. Bear in mind that simulator related attributes already exist in the data model described. Information on the type of the node to be initialized (e.g. station, exit, etc.) is not neutral but closely related to the logic maintained by the simulator.

4.4.1 Embedded JSON schema

With the graph representation model already presented, it is possible to attach properties on nodes and edges, but the schema of these properties is not specified. One of the major concerns during the development of the API was to have a schema describing the data format of the graph representation, not only rendering

the validation of model data against the schema possible, but also enabling users to build dynamically the user interface for any instance updating and formatting of the schema as they wish.

The *class_definition* object is used for the definition of the basic node and edge object types. In that way, it is possible to embed a JSON schema in the API data model. The definitions provided are neutral and can be used to define any model that can be described as a directed graph. Furthermore, the definitions collected under *class_definition* can be used for the synchronization and configuration of the platform, and the GUI. As described in chapter 3, in any use-case scenario, an end user will have to set-up the instance by modifying the *class_definition* object of the all-inclusive JSON file. The JSON schema has already been successfully used to dynamically generate user interfaces, with the best example being the advanced alpacajs form library (`http://alpacajs.org/`).

As we see in listing 4.11, the *class_definition* echelon contains two fundamental objects, the *node* and *edge* definitions. These two basic definitions schemas are inherited by other objects definitions (i.e. "Dream.Edge" and "Dream.Queue") that are instance specific. Generic definitions that are reusable by different object schemas (e.g. a normal distribution definition) are available within *definitions*.

Listing 4.11: *class_definition* example

```
"class_definition": {
  "edge": {+},
  "node": {+},
  "definitions": {+},
  "Dream.Station": {+},
  "Dream.Edge": {+},
  "Dream.Exit": {+},
  "Dream.Buffer": {+},
  "Dream.Resource": {+},
},
```

The node and edge objects definition are provided in Listings 4.12 and 4.13. As explained in section 4.3.1.1, every node is characterized by a "name" property. Additionally, each node has a "_class" property that explicitly defines the class of the SE object that corresponds to that node (i.e. "Dream.Machine", "Dream.Operator", etc). Finally, every node within the workflow editor is given

a set of coordinates ("left" and "top") that reserve a position within the workflow editor GUI panel for that node.

Listing 4.12: generic node schema

```
"node": {
  "description": "Base definition
      for node",
  "properties": {
    "_class": {
      "type": "string"
    },
    "coordinate": {
      "properties": {
        "left": "number",
        "top": "number"
      },
      "type": "object"
    },
    "name": {
      "type": "string"
    }
  },
  "required": [
    "name",
    "_class"
  ],
  "type": "object"
},
```

Listing 4.13: generic edge schema

```
"edge": {
  "description": "Base definition
      for edge",
  "properties": {
    "_class": {
      "type": "string"
    },
    "destination": {
      "type": "string"
    },
    "name": {
      "type": "string"
    },
    "source": {
      "type": "string"
    }
  },
  "required": [
    "_class",
    "source",
    "destination"
  ],
  "type": "object"
},
```

Similarly to generic nodes definition, it is obvious from Listing 4.13 that every edge is characterized by a "name" and a "_class" and two additional properties for the definition of the edge's direction. The "source" and "destination" properties are string objects and are given the *ID*s of the nodes the edge is connecting as values.

Every JSON schema object can have a "required" property which defines the properties of the object that are required so that the JSON object is valid when tested against the JSON schema. For all nodes and edges objects a "_class",

"Source" and "destination" properties are always required for edge objects in order to be valid.

A instance specific edge object definition (e.g. "Dream.Edge" in Listing 4.14) must inherit from the generic *edge* schema. On the generic schema, extra properties can be added by use of the "allOf" reserved JSON schema key. That key requires that all the instance objects of the schema must be valid against all the schemas that are listed within the list object of "allOf". In the example of Listing 4.14, we can see that only one reference is given, pointing to the generic edge definition. The definition can be expanded with properties that will for example be used for rendering the workflow editor UI; different colours or shapes for the edges, or other constraints that can deal with the forwarding of DES entities through that edge.

Listing 4.14: Example of instance specific edge schema

```
1  "Dream.Edge": {
2    "_class": "edge",
3    "allOf": [{"$ref": "#/edge"}],
4    "description": "Connect stations together"
5  },
```

A DES object can be described in terms of JSON schema nomenclature, inheriting from the generic node object definition. For instance, a "Dream.Machine" node that corresponds to an instance of a machine object within the Dream DES repository, has the JSON schema definition provided in Listing 4.15. The "_class" key refers to the GUI workflow editor's node object class, as to distinguish it from the edge class objects. It is used by the DREAM workflow editor for setting-up different instances and for its synchronization. The "css" key and the corresponding value is used for defining the graphic representation of the different DES objects used in each instance. A background image can be defined, a border width and colour can be configured. These keys are not used for validation purposes of the DES data.

Listing 4.15: Example of a simplified machine object schema

```
"Dream.Machine": {
  "_class": "node",
  "description": "station processing items for some time (by a distribution)",
  "css": {
    "backgroundColor": "#fef",
    "backgroundImage": "linear-gradient(to bottom, #fef 0%, #ede 100%)",
    "border": "1px solid #cbc"
  }
  "allOf": [{
      "$ref": "#/node"
    },
    {
      "type": "object",
      "properties": {
        "id": {
          "default": "M",
          "type": "string"
        },
        "name": {"default": "Machine", "type": "string"},
        "processingTime": {
          "$ref": "#/definitions/_dist",
          "default":{"Fixed":{"mean":1}}
        },
        "operationType": {
          "type": "object"
          "properties": {
            "processing": {
              "$ref": "#/definitions/_operationType",
              "description": "Operation type for processing",
              "name": "Processing Operation type"
            }
          }
        }
      }
    }
  }]
}
```

Any node (JSON object) of class Dream.Machine should be valid against the generic node schema, and a schema which contains definitions for an "id", "name", "processingTime", and "operationType". Depending on the different features of an object, more input definitions (properties) may be required. For example an exit station, where movable entities are merely destroyed, does not require a processing times definition. On the other hand, there might be a need for interruption or failure times definitions together with repair time related values, and the corresponding resources used for the relevant actions. Consider the case of a buffer where a capacity property must be defined.

Properties default values are usually characterized by a "default" key, and can be assigned together with the definition of the property. They can be simple string or numerical values or more complex object. In the example of Listing 4.15, the default value of the processing time is "{"Fixed":{"mean":1}}". In any case, the default values must comply with the definition provided, in other words

they must be valid objects against the definition of the schema of the object. At initiation time of the platform, those values are used by the GUI for setting-up an instance with parameter values as defined by the modeller.

4.4.1.1 Definitions

Properties such as processing times may be defined within *definitions* echelon of the *class_definitions* schema. These properties are used by multiple objects and thus, it is convenient to define them once and refer to them by means of "$ref" keyword. There might be definitions for capacity ("_capacity"), failures ("_failureDist"), scheduling rules ("_schedulingRule"), and many others (listing 4.16).

As an example, a generic definition of a time distribution is presented in Listing 4.16. For synchronization purposes of the platform GUI, the modeller may be given various distribution options for the processing times of different nodes (i.e. exponential, normal, etc.). An enumeration ("enum") of distributions is provided to the end user to choose from. The enumeration can be represented in the GUI as a drop down menu. "allOf" keyword dictates that the processing time distribution object should comply with all the schemas listed. The first schema defines the enumeration of distributions (of type string). The second schema of the "allOf" list dictates that any distribution object should be valid against one of the referenced definitions provided under the "oneOf" keyword. For example, "_fixed" has different properties than "_normal" (mean for "fixed" and mean-stdev for "normal"). Default values for such properties are defined by the "default" keyword. The "title" keyword is used by the GUI application of the platform for correctly naming the corresponding input fields. The "description" keyword can be used for documenting purposes as well as for providing intuitive info to the end user of the platform through the GUI.

Listing 4.16: Definition of time distributions example

```
"definitions": {
    "_capacity": {+},
    "_failureDist": {+},
    "_operationType": {+},
    "_schedulingRule": {+},
    "_dist": {
      "allOf": [{
          "properties": {
            "distribution": {
```

```
            "default": "Fixed",
            "enum": ["Fixed", "Normal"],
            "type": "string"
          }
        },
        "type": "object"
      },{
        "oneOf": [
          {"$ref": "#/definitions/distributionTypes/_fixed"},
          {"$ref": "#/definitions/distributionTypes/_normal"}
        ]
    }]
  },
  "distributionTypes": {
    "_fixed": {
      "description": "Fixed",
      "properties": {
        "mean": {
          "default": 0,
          "type": "number"
        }
      },
      "required": ["mean"],
      "title": "Fixed",
      "type": "object"
    },
    "_normal": {
      "description": "Normal",
      "properties": {
        "mean": {
          "default": 0,
          "type": "number"
        },
        "stdev": {
          "default": 0,
          "type": "number"
        }
      },
      "required": ["mean", "stdev"],
      "title": "Normal",
      "type": "object"
    }
  }
}
```

When the time distribution is defined in such a way, the modeller should be able to set graphically the processing times of the nodes that correspond to machines during the design phase. It is possible to modify dynamically the properties of the nodes that are defined following such schemas. On selection of a node (Machine 1), the end user chooses the distribution type according to the modelling needs, as shown in Figure 4.9.

After the modeller defines the processing time distribution parameters, the node JSON object for the corresponding node ("M1") is updated accordingly as presented in Listing 4.17. The resulting JSON object is automatically validated against the schema definition presented in Listing 4.16. Other properties of node objects, such as capacities or scheduling rules can be set in the same way.

Figure 4.9: Time distribution - defining a processing time distribution graphically

```
"M1": {
  "processingTime": {
    "Cauchy": {
      "location": 6,
      "scale": 1
    },
    "distribution": "Cauchy"
  },...
}
```

Listing 4.17: Processing time JSON data corresponding to Fig. 4.9

Such an approach guarantees that the end user can easily and in a graphical way access and modify DES objects' properties. In addition, the requirement for consistency of the DES data is addressed, as the JSON data are generated following the *class_definition* JSON schema. Last but not least, it is not required that the end-user accesses directly the JSON data. He can interface with the raw data model and create or update a DES model through the use of the GUI. The responsibility of defining the input properties to be accessed, populated, and updated, lies upon the developer or core user of the platform as explained in chapter 3. The core user is the one responsible to set the JSON schema following the specifications and requirements of the end-user, as described in the current section.

4.4.2 Configuring the application platform

As explained in chapter 3, one of the requirements from the API data exchange model is the possibility to configure the different instances of the platform in an intuitive way through an all inclusive data file. The proposed approach offers the means to define and configure the different GUI widgets for inserting DES related information as well as presenting simulation results. Furthermore, the information flow or in other words, the sequence of plugins as described in section 3.3, can as well be defined within the data exchange model.

A sample layout of the *application_configuration* dictionary is provided in Listing 4.18. It consists of six objects. "general" is a schema describing the general properties required for the initiation of a simulation run, such as maximum simulation time, the date and time that the simulation clock shall start, the time unit used by the SE. "input" and "output" define the GUI widgets or so called views that should be available for interfacing with the platform. "pre_processing", "post_processing", and "processing_plugin" are used to define the sequence of plugins that should be invoked sequentially in order to analyse and prepare the input (output) DES data before (after) a simulation run.

Listing 4.18: Application configuration configuration layout

```
1  "application_configuration": {
2    "general": {+},
3    "input": {+},
4    "output": {+},
5    "pre_processing": {+},
6    "post_processing": {+},
7    "processing_plugin": {+}
8  }
```

The "general" dictionary contains a JSON schema that is used to define the general DES input information required to initiate a simulation run. As mentioned in the section 4.3.2, these can be "timeUnit" for the time units used by the simulator, "confidenceLevel" for the confidence level of the statistical analysis, and many more depending on the type of analysis requested. An example of "general" schema is given in Listing 4.19. The corresponding GUI view which collects the relevant information is given in Figure 4.10. We can see that for every property of the schema, an input field is set up.

Eventually, the information collected populates the "general" object of the data model presented in Figure 4.2 on page 53. In Figure 4.20 on page 76, we can see the "general" JSON data as collected in Figure 4.10.

```
"general": {
  "properties": {
    "maxSimTime": {
      "default": 100,
      "description": "Length of the
          simulation run",
      "title": "Length of Experiment",
      "type": "number"
    },
    "numberOfReplications": {+},
    "seed": {+},
    "confidenceLevel": {+},
    "timeUnit": {+},
    "currentDate": {+}
  }
},
```

Listing 4.19: Definition of the "general" input

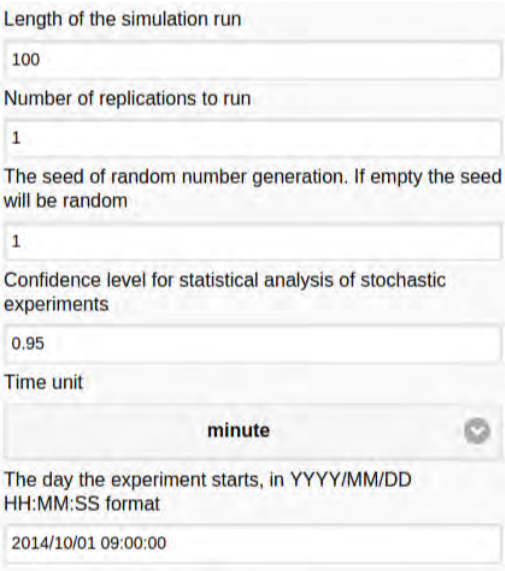


Figure 4.10: General data input - view where input properties defined in 4.19 can be inserted

Listing 4.20: "General" as collected from GUI in Figure 4.10

```
"general": {
  "maxSimTime": 100,
  "numberOfReplications": 1,
  "seed": 1,
  "confidenceLevel": 0.95,
  "timeUnit": "minute",
  "currentDate": "2014/10/01 09:00:00"
},
```

4.4.2.1 Views

Different views or widgets may be available through a graphical interface. They can be used for inserting DES related information to the data model, or for viewing the simulation analysis results. "input" and "output" define the views that should be available for interfacing with the platform. "input" is used for defining the input views and "output" is for the result views.

Input In Listing 4.21, an example of "input" views definition is given. Every view dictionary has a key (i.e. "view", "view_machine_shift_spreadsheet", etc), and different properties such as "gadget", "title", and "type". The "gadget" property is the name of the widget that is invoked. There might be multiple instances of the same gadget, as for example the spreadsheet widget ("Input_viewSpreadsheet"). Though other gadgets must be instantiated only once, as for example the workflow gadget ("Input_viewProductionLine"). The type of all the gadgets is "object_view" to define that the gadget to be instantiated is a view. Different gadgets may be initiated but this is not implemented in this thesis. The title of the view is provided by the property with the same name - "title". These titles can be used for the display of the different gadgets on a panel within the interface (Figure 4.11).

Listing 4.21: "input" views

```
"input": {
  "debug": {+},
  "view": {
    "gadget": "Input_viewProductionLine",
    "title": "Production Line",
    "type": "object_view"
  },
  "view_machine_shift_spreadsheet": {
    "configuration": {
      "columns": [
        {
          "format": "date-time",
          "name": "Date",
          "type": "string"
        },
        {
          "name": "Machine",
          "type": "string"
        },
        {
          "name": "Start",
          "type": "string"
        },
        {
```

```
            "name": "Stop",
            "type": "string"
          }
        ]
      },
      "gadget": "Input_viewSpreadsheet",
      "title": "Machine Shifts Spreadsheet",
      "type": "object_view"
    },
    "view_management": {+},
    "view_repairman_shift_spreadsheet": {+},
    "view_result": {+},
    "view_run_simulation": {+},
    "view_wip_spreadsheet": {+}
  },
```

Several views may also have a configuration provided to the person responsible for the maintenance of the instance. That configuration option comes in the form of a separate dictionary within the view under the keyword "configuration". Currently the only configurable input views are spreadsheets as regards the number, types, and titles of the columns. A similar approach can be used to parametrize other generic GUI gadgets (input views such the workflow editor, output views such as graphs). An example of a configurable view is given in Listing 4.21. The gadget as eventually presented to the end user can be seen in Figure 4.12. That gadget encapsulates, and leads to the "output" views, if they are available.

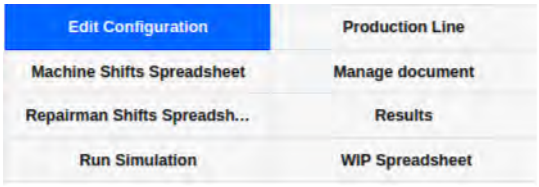


Figure 4.11: Views panel - the panel is updated according to the views configuration given in Listing 4.21

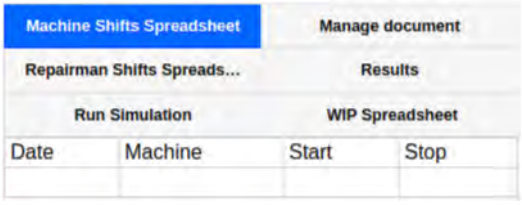


Figure 4.12: Spreadsheet gadget - view is configurable according to the definition in Listing 4.21

Output The output views are defined in an identical way to the "input" definition. As explained in the previous subsection, the "view_result" input-view leads to the different results views on selection. An example "output" configuration dictionary is provided in Listing 4.22. As in the case of "input" views, the output views have a "title", a "type", and a "gadget" property.

The "gadget" property is the name of the GUI gadget to be instantiated. For results presentation, one can make use of a wide arsenal of gadgets, such as spreadsheets, Gantt charts, plot or line graphs (i.e. "Output_viewSpreadsheet", "Output_viewGantt", "Output_viewGraph", etc.). Other output representation gadgets might also be incorporated.

Lastly, there is also a "configuration" for all of the "output" views. Currently, the options available are limited to an "output_id". The property "output_id" relates to a dictionary with keyword equal to "output_id", within the *result* dictionary (see Figure 4.2 on page 53). That dictionary contains DES results prepared in a proper format for the gadgets presenting them graphically. That formatting is performed by post-processing plugins as defined by the "pre_processing" dictionary of the "application_configuration" (see listing 4.18 on page 75).

Listing 4.22: "output" views

```
"output": {
  "view_buffer_state": {
    "configuration": {
      "output_id": "buffer_output"
    },
    "gadget": "Output_viewSpreadsheet",
    "title": "Buffer Statistics",
    "type": "object_view"
  },
  "view_operator_gantt": {+},
  "view_operator_utilization": {+},
  "view_download_spreadsheet": {+},
  "view_exit_results": {+},
  "view_queue_stats": {+},
  "view_station_utilization": {+}
},
```

The relation between the "configuration" property of the different output views and the *result* dictionary is better explained in Listing 4.23 with the description given in the listing.

4.4.2.2 Plugins

As described in section 3.3 on page 43, a number of plugins is invoked serially and prepare the DES data collected within the JSON file prior or post any simulation run. Even the SE is invoked as a plugin performing the simulation on the model data. That sequence of plugins are defined by the "pre_processing", "post_processing", and "processing_plugin" dictionaries of the object "application_configuration". For an example of "post_processing" plugins sequence definition, refer to Listing 4.23. Remark that in that example only one plugin is invoked. Usually the list numbers more than one plugins which are invoked one after the other in a FIFO way.

In Listing 4.24, one can find a simple example of "pre_processing" and "processing_plugin" configuration dictionaries. Similarly to "post_processing" plugins, an "input_id" is provided for some pre-processing plugins. That id refers to keyword identifying the raw data table or data structure that can be found within *input* dictionary. The data is collected and placed there by the corresponding GUI view (e.g. a WIP spreadsheet), or by an other pre-processing plugins that gather the raw data from external data sources. An example on the use of plugins is provided in chapter 5. The "processing_plugin" defines the execution script that invokes the default SE stand alone or combined with any possible optimizers.

In depth knowledge of the platform, and the corresponding features is required for the definition of the plugin execution sequence. High flexibility is achieved by defining the sequence of the plugins in that way. A developer can introduce new scripts performing different kind of processing on newly collected raw data, and the platform will be updated and synchronized simply by updating the data model and specifically the sequence of plugins to be invoked.

Listing 4.23: "output_id" example

```
{
  "application_configuration": {
    "ouput": {
      "view_buffer_state": {
        "gadget": "Output_viewSpreadsheet",
        "configuration": {
          "output_id": "buffer_output",
          "description": "points to a dictionary/list within result dictionary"
        }
      }
    },
    "post_processing": {
      "plugin_list": [{
        "_class": "dream.plugins.BufferLevels",
        "output_id": "buffer_output",
        "description": "reads the raw DES data and prepares a data structure
            for the view designated to read it (view buffer state)"
      }]
    }
  },
  "result": {
    "result_list": [{
      "description": "contains the data structure as prepared by the post-
          processing plugin",
      "buffer_output": [+]
    }]
  },...
}
```

Listing 4.24: Plugins example

```
{
  "application_configuration": {
    "processing_plugin": {
      "_class": "dream.plugins.plugin.DefaultExecutionPlugin",
      "description": "invokes the SE, or optimizers on the SE"
    },
    "pre_processing": {
      "plugin_list": [{
        "_class": "dream.plugins.WIPSpreadsheet.WIPSpreadsheet",
        "input_id": "WIPdata",
        "description": "reads the raw WIP spreadsheet data (tabular) and
            prepares a WIP data structure appropriate for the SE"
      }]
    },...
  },
  "input": {"WIPdata": [+] },...
}
```

4.4.3 Collecting and collating data from various data sources

Valuable information for a DES simulation can be collected from various data sources. As explained in chapter 3, in the approach proposed by the current research, the model is designed using the graphical interface of the platform as a directed graph. DES related information (processing times, capacities, current work in progress, shifts) can be extracted from different simple or complex data sources. The simplest way would be to read the information from spreadsheets. A better and more robust way would be to extract the information required for the DES simulation from an MES or ERP system.

Chapter 3 shows that the platform provides the means to collect data both by reading GUI integrated data sources (GUI spreadsheet, uploaded files) or accessing external DBs by means of the KE tool. Similarly, the API data model should provide the means for the SE to interface with raw data. Through the data model the collected DES data is related and referenced by the DES model (*graph*). As described in the previous section 4.4.2, the data model provides the means for the interfacing and the collected data preparation. We will now see how

the interfacing of that information is performed, and how the data are structured in a meaningful way.

As mentioned in section 4.3.3 on page 62, unprocessed data can be stored by the data model within the *input* object. The *input* object can contain raw data in different formats that can be described in JSON; WIP information, Bill Of Materials, or any other information that cannot be described within the *graph*.

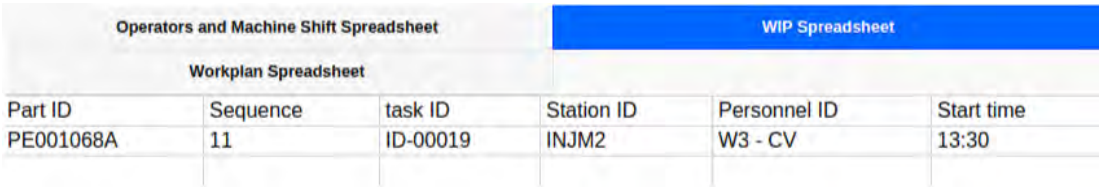


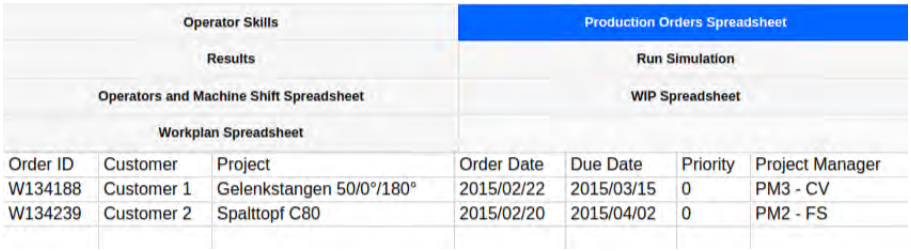
Figure 4.13: WIP example - Work In Progress is defined through the GUI

Considering the WIP information, in Listing 4.24, an array named "WIP-data" can store information concerning the Work In Progress within a job-shop. In Listing 4.8 on page 63, similar information is stored within the array "WIP_spreadsheet". The WIP info can be introduced to the platform via the GUI interface as shown in Figure 4.13. As explained in section 4.3.3 on page 62, different kinds of information can be valuable for the definition of the work in progress; This information is later processed by plugins and eventually can be attached either to *graph* nodes as WIP information, or to a "WIP" dictionary of the *input* object.

Similarly, the moving entities of a job-shop DES model have interconnections between them. The info can be collected through the GUI (ie spreadsheet), or from an external relational DB. In Figures 4.14 and 4.15, we see an example of a production order definition and its corresponding order components.

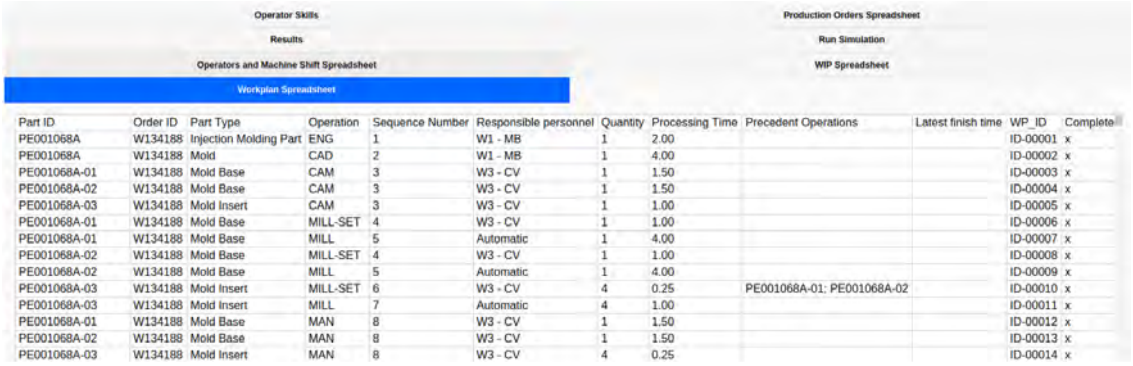
The relevant raw data is stored by the GUI in array format within the *input* object of the API data model as "production_order_spreadsheet" and "work-plan_spreadsheet" (listing 4.25). As explained in section 4.4.2.2 on page 80, a number of plugins process the data and structure them in a SE meaningful format.

The "BOM" is the object that holds the data structures that are identified within the modelled system as moving objects. The different properties of "BOM"



Order ID	Customer	Project	Order Date	Due Date	Priority	Project Manager
W134188	Customer 1	Gelenkstangen 50/0°/180°	2015/02/22	2015/03/15	0	PM3 - CV
W134239	Customer 2	Spalttopf C80	2015/02/20	2015/04/02	0	PM2 - FS

Figure 4.14: Production Orders - Production Orders definitions for a job-shop



Part ID	Order ID	Part Type	Operation	Sequence Number	Responsible personnel	Quantity	Processing Time	Precedent Operations	Latest finish time	WP_ID	Complete
PE001068A	W134188	Injection Molding Part	ENG	1	W1 - MB	1	2.00			ID-00001	x
PE001068A	W134188	Mold	CAD	2	W1 - MB	1	4.00			ID-00002	x
PE001068A-01	W134188	Mold Base	CAM	3	W3 - CV	1	1.50			ID-00003	x
PE001068A-02	W134188	Mold Base	CAM	3	W3 - CV	1	1.50			ID-00004	x
PE001068A-03	W134188	Mold Insert	CAM	3	W3 - CV	1	1.00			ID-00005	x
PE001068A-01	W134188	Mold Base	MILL-SET	4	W3 - CV	1	1.00			ID-00006	x
PE001068A-01	W134188	Mold Base	MILL	5	Automatic	1	4.00			ID-00007	x
PE001068A-02	W134188	Mold Base	MILL-SET	4	W3 - CV	1	1.00			ID-00008	x
PE001068A-02	W134188	Mold Base	MILL	5	Automatic	1	4.00			ID-00009	x
PE001068A-03	W134188	Mold Insert	MILL-SET	6	W3 - CV	4	0.25	PE001068A-01; PE001068A-02		ID-00010	x
PE001068A-03	W134188	Mold Insert	MILL	7	Automatic	4	1.00			ID-00011	x
PE001068A-01	W134188	Mold Base	MAN	8	W3 - CV	1	1.50			ID-00012	x
PE001068A-02	W134188	Mold Base	MAN	8	W3 - CV	1	1.50			ID-00013	x
PE001068A-03	W134188	Mold Insert	MAN	8	W3 - CV	4	0.25			ID-00014	x

Figure 4.15: Work-plan - Order components routing definition

are detailed in section 4.3.3. The responsible plugins manipulate the raw data (e.g. the arrays "workplan_spreadsheet") and format them properly under the "BOM" object of the data model. In Listing 4.9, we can see an example of the transformed data from Figures 4.14 and 4.15.

Listing 4.25: *input* raw data arrays corresponding to Figures 4.14 and 4.15

```
{
  "input": {
    "production_orders_spreadsheet": [[
        "W134188",
        "Customer 1",
        "Gelenkstangen 50/0\u00b0/180\u00b0",
        "2015/02/22",
        "2015/03/15",
        "0",
        "PM3 - CV"
      ],[
        "W134239",
        "Customer 2",
```

```
        "Spalttopf C80",
        "2015/02/20",
        "2015/04/02",
        "0",
        "PM2 - FS"
    ]],
    "workplan_spreadsheet": [+],...
  },...
}
```

4.4.4 DES result data processing

As mentioned in section 4.4.2 on page 74, there are different GUI output views. Each of these views, is responsible for presenting different results to the end user. A view will be responsible to present Gantt charts, an other one will handle buffer utilization levels graphs, etc. The result data must be formatted before handled over data to the output GUI widgets. For that purpose, a series of post-processing plugins process the data before they are handed over to the GUI. The process of formatting the data is described in detail in Chapter 5.

Listing 4.26: Post-processed result data

```
{
  "result": {
    "result_list": [
      {
        "_class": "Dream.Simulation",
        "key": "solution1",
        "score": 0,
        "elementList": [+],
        "order_lateness": {
          "order1": {
            "completionDate": "2015/02/26 17:15",
            "dueDate": "2015/04/02 00:00",
            "delay": -822.75
          },
          "order2": {+}
        }
      },...]
```

```
18   }
19 }
```

The results are selected by the plugins and structured according to the specifications of the GUI widgets. These results are stored in the result objects, under the "result_list" dictionary. In Listing 4.26, an example of the formatted output data is given[1]. In this respect, the data model is expandable and any format (JSON supported structures) satisfying the needs of a platform using the API data model can be supported. For more details refer to Listing 4.23 or to Chapter 5.

4.5 Conclusions

The proposed API data model presented in this chapter satisfies the data exchange interface requirements described in chapter 3. A detailed diagram of the API data model structure is provided in 4.16. The four basic objects of the data model (*general*, *graph*, *input*, and *results*) can cover the data exchange interface needs between a simulation platform and any external modules. General attributes concerning a simulation experiment can be described in *general*. The topology of a DES model in terms of directed graph nomenclature is included in the *graph* object. Extra information about movable DES entities and relationships between them, as well as initial conditions can be described within the *input* object, under "BOM" and "WIP". Eventually the simulations results are returned by the simulator using the *result* object. The same layout can be used for different DES simulators and platforms with the use of appropriate parsers.

Extra definitions and specifications that have to do with the DES platform synchronization are also included in the data model. *Class_definition* and *application_configuration* objects dictate the way the platform is synchronized, DES related information is collected, parsed, and pre- or post-processed. The instance of the GUI is created following the *application_configuration* and *class_definition*

[1]According to subsection 4.4.2.1, the view that is configured to read the output data with id "order_lateness", will select the result data with the corresponding key. The format of the data structure (list, dictionary, or complex objects) is defined by the GUI gadget that reads the output data with the corresponding key.

Figure 4.16: Extended JSON API structure- Detailed structure of the JSON API data model

dictionaries instructions. The *graph* is generated in the GUI workflow editor according to the definitions of *class_definition*. Raw data collected by external data sources are formatted by a sequence of parsers described by *application_configuration* and stored in a structured way under *input*. The simulation results are eventually formulated according to the modellers needs again by a series of parsers-plugins as instructed by *application_configuration* and stored under *result* object. The information flow as presented in Figure 3.5 can be expanded to Figure 4.1 according to the explanation given in section 4.2 on page 51.

Class_definition dictionary can be seen as an embedded schema for the creation of valid DES data. The definitions provided by the schema are neutral and can be used to define models that can be described as directed graphs. In addition, the schema is expandable and can incorporate further definitions that are not currently incorporated, for example, moving entities' definitions and properties descriptions are not included, but the schema can be easily extended. The requirement for DES data consistency is addressed, as the JSON data are generated following the schema.

Additionally, the embedded JSON schema enables the definition of the input requirements of entities or process of DES models. The data model can also cater for the interfacing of different modules and plugins of the platform based on referencing. In this way the synchronization of the platform can be achieved, and the interoperability between the existing modules of the platform or any future module is guaranteed.

The architecture described in Figure 4.1 guarantees that an end user can easily and in a graphical way access and modify DES object's properties, similarly to the features provided by COTS DES software. The end user is not required to directly interface with the JSON data. The responsibility of configuring a platform instance by deciding on the JSON data configuration lies upon the developer/core user of the platform. The expandability of the API data model offers the opportunity for extra features to be introduced.

5

Validation of JSON API

The JSON API introduced in chapter 4 was validated using three pilot cases which were cases within the DREAM project. Different use scenarios were implemented in different pilot case studies. Each of the case studies posed different requirements to the platform, and thus to the Data Exchange data model. The expandability of JSON API data model is tested against the varying data exchange requirements and needs.

Three different use case scenarios are introduced and briefly described. Then one of the case studies is presented in more detail, and the required configuration of the platform instance is discussed. The proposed solution is then described from the JSON API point of view. Examples of the information flow, the data configuration, and data pre/pro-processing are given.

5.1 Problem statement

In order to present the capabilities of the JSON API data model, a case study is detailed in the current chapter. Initially, three different case studies implemented within the scope of DREAM project are briefly introduced. Each one of the case studies introduces different requirements and tests the limits of the JSON API data model in different ways. Every case study is based upon a solution for a real use case scenario within a company.

The scope of the current chapter is to give a brief example of how data is exchanged between the different modules within the DREAM platform. In addi-

tion, we aim to present how a platform is configured according to the needs of a specific case scenarios. The information flow as described in chapter 3, is adopted to the needs of the case studies examined.

5.1.1 Description of CASE1 Case Study

CASE1 is a Turkish organisation involved in designing and manufacturing of cotton gin machinery and textile machinery. The company has many collaborating agencies/offices in numerous countries all over the world. CASE1 group consists of three departments: (i) Textile Machinery; (ii) Cotton Gin Machinery, and (iii) Import & Export.

Textile and ginning machine manufacturing takes place in separate but linked facilities. The number of employees ranges from 200 to 240 in high season. Half of the manpower is allocated to ginning machinery manufacturing and 40% to the textile machinery production line. The rest of the company's manpower perform administrative tasks. One of the core competencies of the company is considered to be the flexibility in its production. This leads to a great product diversity. Products can be defined as machines or lines depending on the scale of the product:

- A machine is a self-contained element generally performing a singular function,
- A line is a set of machines linked in order to fulfil a specific output.

It is not uncommon for CASE1 to receive an order for a full line, which is then built in its entirety at the company and tested to run against the costumer's requirements, before being dispatched to the client facility. There are standard machines or lines that CASE1 offers, but the vast majority of the orders are customised according to the needs of the clients. So a make-to-order approach is followed. After an order is accepted, the first material (steel, electric circuits etc.) is produced, and then they design and build the ordered machines.

The textile and ginning machining facilities have similar functionalities. Nevertheless, textile machinery manufacturing is considered as the focal point due to the textile market decrease in Turkey. This decrease results in a need to gain a greater proportion of the international market. In this environment of intense

international competition, a decision support solution is considered to be more beneficial in terms of impact on the key business metrics.

5.1.1.1 Production Processes

In this section, the basic decision and production processes are outlined, in order to provide a better understanding of the production management issues faced by CASE1. For ease of exposition, these processes can be considered to run serially, i.e. managing the incoming order and planning production, then implementing the plan.

After an order is accepted, the production process is initiated. This is when a decision is made on how the order will be handled (e.g. onsite production, outsourcing etc.) and what material is going to be needed. As mentioned in the previous section, CASE2 follows a make-to-order approach; this phase also involves R&D design operations for tailoring the products according to customer specifications.

As soon as the project plan is ready, the work orders are sent to the subsequent departments. These include departments responsible for procurement and preparation of electronic parts (e.g., control panels). Production is initiated once all the parts of an work order are acquired. Purchasing deals with acquisition of the necessary materials according to the project plan. These include raw materials and outsourced, produced parts required for the manufacturing of parts of a project. There is an inventory kept for common parts. Note that there is a threshold level of the inventory. The remainder of the production process can be summarised as:

- Production of the framework of a machinery,
- Machining the internal parts,
- Preparation of the electronics parts,
- Painting and drying,
- Assembly, testing and packaging.

Production of the machinery body-framework involves sheet metal forming for the main body parts and welding them in one assembly. The formed products are dyed in the painting department and after that it enters the furnace in order to dry. The oven that is used is considered small (8×3m) and it can fit a small

number of machines inside it at a time, depending of course on the size of the machines. The fully constructed and painted products (either on-site or by sub-contractor) are assembled along with the electrical panels and cables. The whole production line is constructed and at CASE2 company's facilities. Then the full product gets dispatched and shipped to the customer.

5.1.1.2 Pilot Case

The pilot case is centred at the CASE1 textile order management. The identified key metric is the lead-time that needs to be evaluated and minimised. The simulation model of the textile production line should mimic the system, taking into account its variability, and provide the estimated lead-time for all the orders in progress. The expected users to interact with the model are the ones responsible for the decisions mentioned above (higher management of the organization).

In this approach, resources mainly refer to two different types, man-hours and space for specific assembly operations. Projects that are assembled occupy space that is a limited resource. Given the fact that the production lines the organisation manufactures may have several shapes, the task of allocating the space optimally is also a complex matter. It is out of scope for the designed approach to solve this issue, but it is required to provide the floor space requirement of each project as input in order to be handled by the simulation engine.

The resources can thus be described as capacity stations. Every capacity station handles a limited floor space as resource assigned to it. This attribute is critical for assembly stations, where floor space limitations are crucial A capacity station also shares worker resources (man-hours) with other stations. Priorities are also assigned to every station, as certain tasks may have priority against others within the shop-floor.

The movable entities through the manufacturing system are production orders. For every order the end user is requested to provide the production step sequence, the assembly space requested for every step, the capacity requirement (in man-hours), and finally an order date together with a due date. WIP information is also of great importance for correct calculation of lead times. as well as the available capacity of every station versus time.

The end user, must be in position to create new simulation models or modify existing ones, providing all the required information for the instantiation of a simulation run. The info required as input for the simulation model, defines the instance configuration and the DES API data model important properties.

5.1.2 Description of CASE2 Case Study

The company develops, manufacturers and markets medical devices across a range of application areas including but not limited to cardiology, vascular surgery and endoscopy. Worldwide, the organisation employs approximately 24,000 people with a sales force in over 40 countries. There are 15 manufacturing facilities across the globe.

The organisation's site in total is comprised of 30 acres with 195,000 sq. ft. of manufacturing related floor space of which 74,000 sq. ft. is a controlled clean-room environment compliant to relevant standards such as Federal Standard 209 of the USA. There are 7 separate clean room areas, from which products are fed to a single on-site warehouse.

The site operates a 3-shift operation and the vast majority of its products rely on skilled and semi-skilled manual operations and could be classed as labour-intensive, and medium skill with technological intensity. The site employs 1,688 individuals in total and the production workforce is generally split between permanent staff and those on temporary contracts. Those involved directly in production are "Product Builders" or PBs.

The pilot case site currently supports 63 product lines and 857 end product codes. The production can be described as having a diverse product range. Total output in 2012 was 5.9m units across 4 of the corporate product divisions: (i) Peripheral Intervention, (ii) Endoscopy, (iii) Interventional Cardiology, and (iv) Urology. Endoscopy comprises 20% of revenue from the site and is the focal point of the case study.

5.1.2.1 Production Processes

The company's process steps are in there majority manual in nature and require detailed training due to the critical techniques required to build medical devices.

The company is governed by Food and Drug Administration (FDA) and similar worldwide bodies so attention to detail and traceability is critical. Production volumes can vary greatly depending on the production line and product, from a few hundred units per month up to 140,000 units per month at the site, and over time does not tend to vary too much from month to month. The majority of lines would only vary in volume by +/- 10% from month to month.

The manual nature of production at the facility leads to a targeted cross training flexibility programme of support such that:

- Each permanent member of staff gets trained on 3 different stations (within that line);
- Each different station has 3 staff members trained on it;
- Each temporary contact worker gets trained on a single station.

It takes approximately 2 weeks to train a PB onto a station. A PB will stay with their primary station, not rotate within the 3 upon which they are trained; they move when an event occurs that requires this movement. When a PB is absent there is no ranking from selecting the back-up PB, it is at the discretion of the team leader. A team leader position is that of line supervisor.

The team leader deals with operational decisions such as headcount allocation, machine utilisation, management of WIP buffer levels at stages along the production line and sequencing plans for WIP through a station and overall line. Before a shift, the team leader makes operational decisions and also reviews status on an hourly basis. Progress is additionally reviewed by the value stream manager twice daily against the target determined by the build plan. The company reviews the following metrics on a monthly basis.

- Output,
- Efficiency (or productivity),
- Cycle Time,
- Line Attainment (daily meeting of production output plan)

In parallel with the production hierarchy but outside of its line-management, each area also have a “core team”, which is a shared services team made up of supporting functions including: (i) Manufacturing engineering, (ii) Quality

engineering, (iii) Training, and (iv) Business Process Improvement (BPI), also known as Industrial Engineering.

Production on-site is comprised of 9 shifts per week, with 2 per day except Friday where there is only one.

5.1.2.2 Pilot Case

The pilot case focuses on one production line in one of the clean room areas. In each clean room, there are multiple lines. In the clean room of the pilot line, other lines also operate. Even though the products fabricated in the pilot line have dimensional variations, the pilot line can be considered product dedicated. This means the production flow is the same for the all products. The product in question is a medical application item designed to be used in operating theatres. It comprises of two parts; a 1.8m flexible shaft, and a balloon capable of inflating within a human vein. The shaft means that the product is quite long and flexible and for much of its processing it is kept straight to full length. This has an impact on the product storage throughout the line. Items are produced in batches of 100 units. In Figure 5.1, one can see the process flow diagram of the pilot line.

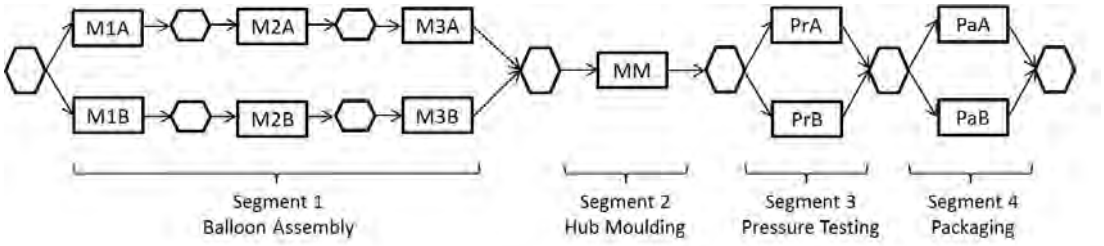


Figure 5.1: CASE2 Process stages - Schematic layout of the pilot line

The line can be considered as an assembly line and different tasks can be assigned to each station for balancing purposes. The process flow consists of four main segments: (i) Balloon assembly, (ii) Hub moulding, (iii) Unit pressure testing, and (iv) Unit packaging. The first segment comprises two serial lines that operate in parallel, each having three consecutive stations across which a production batch can be distributed. A production batch is generally divided into several sub-batches which are passed to the downstream buffer once completed. Before leaving segment 1 a batch must be complete.

As happens for most of the lines in the plant, the process is labour intensive; each station requires one or more operators. The presence of more than one operator per station is mandatory in the last segment (e.g. packaging) where three operators are required to run the two parallel stations. In some sections, the presence of a number of operators greater than the number of stations enables processing time reductions whenever work division is possible. As an example, in segment 3, unit labelling and pressure tests can be carried out separately; hence, the presence of an additional operator at one of the two stations would help speed up the process.

The production is carried out based on two daily shifts running on weekdays; in order to facilitate Work-In-Process (WIP) balancing, production capacity is suitably adjusted at different shifts. For example, with segment 2 the bottleneck in the line, one of the sub-lines in segment 1 is shut down during the evening shift in order to reduce the WIP built up in segment 2 during the day shift.

This case study uses different data sources to record operational information. Raw data from the production line are recorded in batch level on the local CAM-STAR MES (`http://www.camstar.com/`) platform servers and transferred every 3-6 minutes to a local SAP server. Information about the process story in the line for each batch is logged and stored. The records contain different information about the container name, the product and description name, the start and finish time in each station along with the station name, the container quantity during the processing in the line and finally the employee name, which is the name of the employee performs the operation. This information is crucial for the instantiation of a simulation model, as well as for validation purposes.

The data collected by the MES platform needs to be introduced to the DES simulation platform in an automated way due to the volume of the information. This is performed by the KE tool. The KE tool is invoked as a pre-processing plugin as detail in chapter 4. A number of plugins further processes the DES related data before a simulation run is initiated.

5.1.3 Description of CASE3 Case Study

CASE3 is a small an innovative rapid prototyping/rapid tooling enterprise. Since its establishment in 1998, the company has grown into a contract manufacturing service bureau with a wide range of innovative manufacturing technologies for early models and prototypes in different phases up to small series production. It offers the whole prototyping process chain including: Stereolithography, Laser Sintering, Vacuum Casting, Metal Casting, Rapid Tooling, and Injection Moulding. The main customers can be found in the automotive field but also electronics, house-ware, toys, medical and mechanical engineering.

The main products that they offer are Rapid Prototyping, Rapid Tooling, Metal Casting, HSC (5-axis) and Injection Moulding. The organisational structure has been separated to three different manufacturing departments, namely: (i) Rapid Prototyping Department, (ii) Rapid Tooling Department, and (iii) Injection Moulding Department.

The company employees a small number of people, which makes the whole system extremely flexible. The skills-set of the employees allows nearly all of them to work in each position/technology. The level of education and training activities are monitored in the course of the organisation's quality management system (Tricker 2014). The company has managed to gain a strong position in the market owing to the very high quality standards and the absolute adherence to delivery dates and customer requirements.

5.1.3.1 Production Processes

The company receives orders for injection moulded parts from customers with various needs. The orders vary significantly in size since they can range in volume. They may extend from one-off prototypes to reasonably extensive pre-production series.

The injection moulding process in CASE3 consists mainly of two sub-processes; the production of moulds, and the production of the actual plastic parts. Moulds are made of aluminium and regardless of the part types or the order volume the process of mould creation or tooling remains nearly unchanged. The produced

moulds are suitable for the prototyping. On the other hand though, they degrade quicker through use and are not suitable for large-scale manufacture.

In order to create the mould, the ordered part drawings are used to create complementary mould drawing. A mould drawing can be considered basically the inverse of the desired output part. Design data has to be introduced to the machines for the processing to take place. Then the moulds are produced by Milling and possibly Electrical Discharge Machining (EDM). The simplest procedure my require a single session on the milling machine, but depending on the complexity of the mould there might be back and forth transfers between the EDM and the Milling units along with manual material manipulation. EDM is necessary for specific geometries, such as internal angles with pointed corners that milling cannot create, and surface finishes of various quality grades. Eventually, the mould unit is ready for mounting in the machine to produce plastic parts. Two injection moulding machines are currently available at the premisses of CASE3; every machine has different specifications and capabilities. They provide the potentiality to automatically scheduled jobs so that they can be run overnight without human supervision.

5.1.3.2 Pilot Case

Mould manufacturing has been chosen as the focal point of the CASE3 pilot case. It is a sub-process of the plastic injection moulding business, and it has been identified that the company would desire to enhance it with a more sophisticated engineering approach.

The process commences with the arrival of an order. The arrivals of orders are random but there is some periodicity of the demand involved. In addition, different orders may provide a long time horizon while others require a fast delivery, which of course induces more turbulence to planning. The next step is an informal in person consultation with the staff to identify current workloads and particular expertise required for the project etc. These meetings lock in the decision of whether to proceed with the project as proposed. Additionally, in the case of a positive decision for production, a delivery date is defined.

The different stages of the whole process are given in Figure 5.2. The processing of an order starts with CAD design that is conducted once for the whole order. CAD has to be completed in order for CAM to start. Opposite to CAD, CAM design is performed at part level. For a part to start its manufacturing its CAM design should be finished. As one can observe in Figure 5.2, it is the second stage which consists of the manufacturing of mould parts where the operations (Milling, EDM, Drilling, Turning) can be interwoven. Precedence constraints may also be introduced at this level. For example, the milling of one part must take place first so that it is used in the milling of another. Finally, after all parts are manufactured, the mould is first assembled (Mould Assembly) and then used at the Injection Moulding machine where the prototyped parts are manufactured.

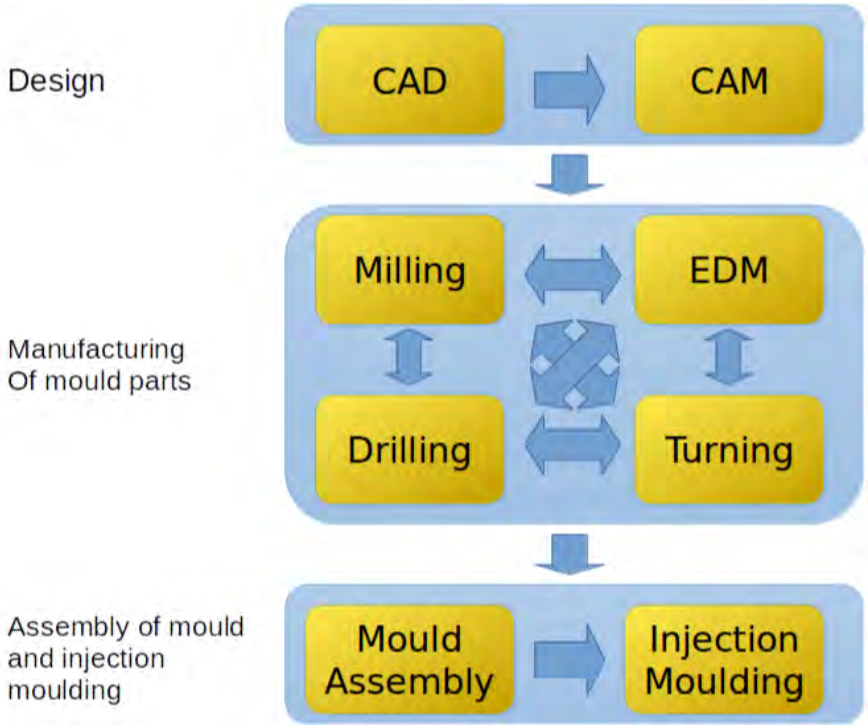


Figure 5.2: CASE3 Process stages - the injection moulding process stages

One single employee is assigned the role of the project manager who undertakes part of the workload and manages the process as he considers best. Though, for every single step of the production sequence of a part, a different employee might be assigned to perform the task. The final process though depends eventu-

ally on the current workload of every staff member, and the original assignments act more as a guideline. Moreover, the project managers make decisions for the projects that they are responsible for individually. The overall approach makes it very hard for the organisation to achieve a holistic approach on project scheduling. It has been identified by the management of the company, that in order to grow the business, a change of operations planning will be required.

5.1.3.3 Requirements

In this work, the focus lies on the pilot case of CASE3, and thus in this section, we will further detail the requirements posed by that pilot case on the configuration of the JSON API data model.

Model Design: As described in chapter 3, it is required that the modeller is enabled to design models in a graphical way similar to what COTS DES solutions offer. The attempt design and model a job-shop similar to the one presented in Figure 5.2, introduces some limitations. Moving entities (order parts and components) can move and be processed within the system in different manners and with varying routes. In other words, the routing within the model cannot be uniquely defined. For every step of a component's route, the processing time, and other properties related to the DES simulation, as for example the responsible personnel to perform every step's task, differ from component to component.

Additionally, components that belong to the same parent order may be interrelated, and their processing may not be allowed to proceed before a sibling component has finished its processing in an other station. This information should be supplied by the end user, together with other information related to the different steps of a order's routing.

Data Collection: The DES information describing the processes to be modelled should either be provided in a graphical way via the workflow editor (see section 4.3.1) and other GUI widgets (e.g. spreadsheets), or collected from external data sources (e.g. ERPs, MES systems). In either way, the information collected should be pre-processed in a consistent way, before the DES model is populated with that info and sent to the simulation engine. For that purpose,

there exist plugins that connect to the different data sources and collect the data in raw format. Other plugins process the collected information (from whatever data source) in a consistent way, and prepare the DES model following the requirements posed by the pilot case production processes.

Results Evaluation: Eventually, the results should be retrieved from the simulation engine in order to be evaluated by the end user. It is requested that concise and meaningful charts are presented to the users, in order to support them with the decision making throughout the life-cycle of an order. For the purpose of scheduling the workload, assigning tasks to resources, and preventing bottlenecks, Gantt or resources utilization level charts can be used. With that scope in mind, the DES results should be post-processed in order to format the data according to the needs of the end users.

5.2 Configuration

As explained in section 4.4.2 of the previous chapter, the DES platform can be configured by use of the *application_configuration* object of the JSON data model. In the current section, we are going to examine what views are needed to cover the requirements of CASE3 pilot case, as well as the sequence of the plugins that pre-process the raw DES data. The way that the data collected are preprocessed and parsed into the JSON API data model is detailed in section 5.4

The post-processing plugins are also going to be discussed briefly. More detailed information on the post-processing plugins will be provided in section 5.5. In Appendix D, an extended JSON file containing all the required information for the initiation of the CASE3 instance can be found.

Finally, we are going to see the object classes required by the SE in order to model and simulate the CASE3 job-shop. The *class_definition* dictionary defines the corresponding nodes and edges and their properties.

5.2.1 Views

It is defined as a platform specification for the CASE3 pilot case, that the modeller can design a DES model or modify an existing one. Further input views requirements consist of;

- The operators skill-set should be defined within a GUI spreadsheet. In other words, the stations that can each employee can operate should be entered by the end user in a GUI spreadsheet

- The production orders that are currently in the system, as well as the newly introduced orders, that must be considered for processing are entered by the end user in a GUI spreadsheet. It is important that the end user specifies the priority of the order, the due date defined by the client, the order date, as these properties are used by the simulation engine's scheduling rules defined by the modeller or the optimization engine.

- The work-plan for every order and each of its component is specified in a different GUI spreadsheet. Every component should follow its own route, and for every step a task-id is defined. For every task, a responsible personnel should be defined. In addition, processing times together processing types (automatic or manual) need to be assigned. It is easier for the end user to define different tasks for every different action performed on a station-machine. For the simulation, the steps that are performed on the same station can be conglomerated on single step, and the different actions are distinguished into set-up and normal processing. We will see how that shift is performed in the plugins section.

- Work In Process (WIP) must also be defined by the user, or extracted by an external data source automatically. As a proof of concept, the platform integrates a GUI spreadsheet for the WIP on the job-shop. For every action that is currently being performed, the SE needs information on the id of the WIP components, the id of the tasks currently performed, and the stations that the processing takes places. Furthermore, the id of the operator performing any task is needed, together with the time that every

action was initiated, in order to calculate the remaining processing time. This calculation is performed by the corresponding pre-processing plugins.

- For reliable decision support, it is crucial that the operator and machine shift conditions and events are reproduced by the SE. A default shift pattern is considered and implemented by the corresponding pre-processing plugin (as we will see in section 5.4). Though, exceptions may be defined by the end users (on case of absenteeism or a machine break down). This information is introduced to the platform by the relevant GUI spreadsheet. Thus, shift start, stop, breaks, as well as the exception date should be inserted by the end user.

In Listing 5.1, one can see a snippet of the *configuration* dictionary that lists the input views defined for the CASE3 pilot case instance. The views that are pilot case specific are five in number. The above listed requirements are satisfied by the following views:

(i) "view_operator_skills_spreadsheet",
(ii) "view_production_orders_spreadsheet",
(iii) "view_workplan_spreadsheet",
(iv) "view_wip_spreadsheet", and
(v) "view_shift_spreadsheet".

The remaining views are not instance specific and are used for the configuration of the default GUI widgets.

Listing 5.1: CASE3 pilot case input views

```
"input": {
  "view": {
    "gadget": "Input_viewProductionLine",
    "title": "Production Line",
    "type": "object_view"
  },
  "view_operator_skills_spreadsheet": {
    "configuration": {"columns": [+]},
    "gadget": "Input_viewSpreadsheet",
    "title": "Operator Skills",
    "type": "object_view"
  },
```

```
13      "view_result": {+},
14      "view_management": {+},
15      "view_production_orders_spreadsheet": {+},
16      "view_run_simulation": {+},
17      "view_shift_spreadsheet": {+},
18      "view_wip_spreadsheet": {+},
19      "view_workplan_spreadsheet": {+}
20    },
```

The "view" view refers to the work-flow editor. "view_management" is a view where end users can export the JSON API file and download it to their computer, or delete the current instance from the PCs. The view "view_run_simulation" allows the user to perform a simulation run and set different *general* parameters as explained in section 4.4.2 (see Figure 4.10 on page 76). In the pilot case of CASE3, the properties that are of interest are: (i) current date, (ii) maximum simulation time, (iii) process timeout, (iv) time unit, (v) seed (for the random number generator of the SE), (vi) number of solutions (return by the optimizer), (vii) number of replications, (viii) number of generations (for the optimizer) (ix) number of ants per generation (for the ant colony optimization routine used by the SE). For a detailed description of each property refer to Appendix B.

There, is also a "view_result" view that links to the output views, and presents to the end user a comparative view between the different results returned by the SE. That dictionary contains an "output_id" property that refers to output results. It behaves in the same way as the output views. For more details refer to section 4.4.2.1 and Listing 4.23 on page 81.

In Listing 5.2, we can see the output views of the CASE3 pilot instance. The requirements which led to the specific selection of output views, consists of;

- In order to make the proper decisions, the management of the company needs access to graphical charts that present the routing of the components, as well as the assignment of the companies resources (i.e. employees), based on optimized solutions.
- A full schedule of the different tasks in tabular format, as calculated by the SE, is also required for reference and debugging.

- The utilization of the operators and the stations is also requested, in order to identify the bottlenecks of the job-shop.

These requirements are satisfied by the views listed under "output" dictionary of *application_configuration*; these views are:

(i) "view_component_gantt",
(ii) "view_operator_gantt",
(iii) "view_component_schedule_spreadsheet",
(iv) "view_station_utilization", and
(v) "view_operator_utilization".

Listing 5.2: CASE3 pilot case output views

```
"output": {
  "view_component_gantt": {
    "configuration": {
      "output_id": "component_gantt"
    },
    "gadget": "Output_viewGantt",
    "title": "Orders Gantt",
    "type": "object_view"
  },
  "view_component_schedule_spreadsheet": {+},
  "view_operator_gantt": {+},
  "view_operator_utilization": {+},
  "view_station_utilization": {+}
},
```

5.2.2 Plugins

Pre-processing: The DES related information collected by the GUI, or other external data sources should be processed and prepared in a format that can be used and parsed by the SE. For that purpose, there exists a list of plugins that process raw or structured DES data. Each one of them performs a certain task, which is described by their titles. The plugins that directly access raw data, and perform some initial processing are usually assigned an "input_id". That id refers to the keyword of the input data within the *input* dictionary, as explained

in chapter 4. Other plugins process already structured data, and thus have no need of an "input_id" to identify the data they need to analyse. All of the plugins inherit from the same generic plugin class, as we will discuss in section 5.4.

In Listing 5.3, one can observe the long list of pre-processing plugins. The list includes plugins for reading raw data from input spreadsheets in tabular format ("ReadJSSkills", "ReadJSOrders", "ReadJSWorkPlan", "ReadJSWIP"). These plugins may also perform some analysis on the data before structuring them according to the DES API data model. Note that in case that the data source is not the GUI, but an external data source, these plugins would have to be replaced by others with the capability to connect to these external data bases. An other option is to add to the list extra plugins extracting the data from the external sources. The same plugins will then structure the data within *input* in tabular format, with keywords that point to the "input_id"s of the already existing plugins.

Listing 5.3: CASE3 pilot case pre-processing plugins

```
"pre_processing": {
  "plugin_list": [{
      "_class": "dream.plugins.ReadJSSkills.ReadJSSkills",
      "input_id": "operator_skill_spreadsheet"
    },{
      "_class": "dream.plugins.ReadJSSkillsToStations.ReadJSSkillsToStations",
      "input_id": ""
    },{
      "_class": "dream.plugins.ReadJSOrders.ReadJSOrders",
      "input_id": "production_orders_spreadsheet"
    },{
      "_class": "dream.plugins.ReadJSWorkPlan.ReadJSWorkPlan",
      "input_id": "workplan_spreadsheet"
    },{
      "_class": "dream.plugins.ReadJSWIP.ReadJSWIP",
      "input_id": "wip_spreadsheet"
    },{
      "_class": "dream.plugins.MergeSteps.MergeSteps",
      "input_id": ""
    },{
      "_class": "dream.plugins.SplitRoute.SplitRoute",
      "input_id": ""
    },{
      "_class": "dream.plugins.ReadJSCompleted.ReadJSCompleted",
      "input_id": ""
    },{
      "_class": "dream.plugins.UpdateStationList.UpdateStationList",
      "input_id": ""
    },{
      "_class": "dream.plugins.InsertQueues.InsertQueues",
      "input_id": ""
    },{
      "_class": "dream.plugins.UpdateWIP.UpdateWIP",
      "input_id": ""
    },{
      "_class": "dream.plugins.InsertWIP.InsertWIP",
      "input_id": ""
```

```
38      },{
39        "_class": "dream.plugins.ReadJSShifts.ReadJSShifts",
40        "input_id": "shift_spreadsheet"
41    }]
42  },
```

The rest of the plugins (i.e. "insertWIP", "MergeSteps", "UpdateStationList", etc.) perform analysis on and format already structured data. It is possible that some of the plugins perform a set of tasks and not a single one, but it is advisable that the different tasks are distinguished for maintainability. For example, the "insertWIP" plugin, reads the WIP already extracted and formatted within the "BOM" object by the "ReadJSWIP" plugin, and updates the WIP of each node (buffers and stations). The "MergeSteps" plugin merges the tasks that refer to one single operation but are split in several subtasks (load, set-up, processing; see section 5.2.1). More information is available in section 5.4.

Processing plugin: As explained in the requirements chapter 3, the SE engine is also invoked by a plugin. An execution plugin is a script (class) that executes the simulation scenario using the ManPy DES library. There exists a default execution plugin, which calls the SE a single time. An exemplar of the default execution plugin is available in Listing 5.4. It performs a single run by invoking the simulation engine (method *runOneScenario*). This plugin can return a "score" for its solution that can be used in optimization. In the CASE3 pilot case, the default execution plugin is replaced by a plugin that inherits from the default one, and performs multiple simulation runs for optimization purposes. Ant colony optimization is used, see "dream.plugins.JobShopACO.JobShopACO", Listing 5.5. For details see the Appendix C.

Listing 5.4: Default execution pluging

```
1 class ExecutionPlugin(Plugin):
2   """Plugin to handle the execution of multiple simulation runs.
3   """
4   @staticmethod
5   def runOneScenario(data):
6     """default method for running one scenario"""
7     return json.loads(simulate_line_json(input_data=json.dumps(data)))
8   def run(self, data):
9     """General execution plugin."""
```

```
10      raise NotImplementedError
11  class DefaultExecutionPlugin(ExecutionPlugin):
12    """Default Execution Plugin just executes one scenario."""
13    def run(self, data):
14      """Run simulation and return result to the GUI."""
15      data["result"]["result_list"] = self.runOneScenario(data)['result']['
            result_list']
16      data["result"]["result_list"][-1]["score"] = 0
17      data["result"]["result_list"][-1]["key"] = "default"
18      return data
```

Post-processing: In Listing 5.5, we can also see the list of post-processing plugins. The post processing plugins are invoked after the scenario has been executed on the SE. According to the requirements (chapter 3), these plugins calculate various key performance indicators such as the number of entities produced per day, the utilisation ratios of available resources etc.. The titles of the plugins are indicative of the tasks they perform; (i) "PostProcessOrderLateness", (ii) "JSStationUtilization", (iii) "JSOperatorUtilization", (iv) "JSOperatorGantt", (v) "JSComponentGantt", and (vi) "JSComponentTabSchedule".

Listing 5.5: CASE3 pilot case post-processing plugins

```
1  "processing_plugin": {
2    "_class": "dream.plugins.JobShopACO.JobShopACO",
3    "input_id": "Simulation"
4  },
5  "post_processing": {
6    "description": "",
7    "plugin_list": [{
8        "_class": "dream.plugins.PostProcessOrderLateness.
             PostProcessOrderLateness",
9        "output_id": "order_lateness"
10     },{
11       "_class": "dream.plugins.JSStationUtilization.JSStationUtilization",
12       "family": "Server",
13       "output_id": "station_utilization"
14     },{
15       "_class": "dream.plugins.JSOperatorUtilization.JSOperatorUtilization",
16       "output_id": "operator_utilization"
```

```
    },{
      "_class": "dream.plugins.JSOperatorGantt.JSOperatorGantt",
      "output_id": "operator_gantt"
    },{
      "_class": "dream.plugins.JSComponentGantt.JSComponentGantt",
      "output_id": "component_gantt"
    },{
      "_class": "dream.plugins.JSComponentTabSchedule.JSComponentTabSchedule",
      "output_id": "component_schedule"
  }]
},
```

Post processing plugins can also perform data preparation for visualization. For example in the CASE3 pilot case, the SE outputs the suggested route for each of the order components, and a post processing plugin ("JSComponentGantt") reformats the data to match the format expected by a GANTT visualisation widget or any other chart widget (GUI). In order to achieve communication and synchronization between the GUI widgets and the post-processing plugins, post-processing plugins are assigned an "output_id" referring to a keyword under every solution as explained in detail in section 4.4.4. Examples of post-processing plugins can be found in section 5.5.

5.2.3 DES object Classes

As we saw already in chapter 4, the *class_definition* object is used for the definition of the basic node and edge object types. In that way, it is rendered possible to embed a JSON schema in the API data model. The *class_definition* echelon contains two fundamental objects, the *node* and *edge* definitions. These two basic definitions schemas are inherited by other objects definitions (i.e. "Dream.Operator" and "Dream.Queue") that are instance specific.

Listing 5.6: CASE3 instance object classes

```
"class_definition": {
  "edge": {+},
  "node": {+},
  "Dream.ConditionalBuffer": {
    "_class": "node",
    "allOf": [+],
    "css": {+},
    "description": "A buffer where entities can be hold until the next station is ready to process them and a certain
        condition is met",
```

```
9       "name": "Queue"
10    },
11    "Dream.Edge": {
12      "_class": "edge",
13      "allOf": [+],
14      "description": "Connect stations together"
15    },
16    "Dream.ExitJobShop": {
17      "_class": "node",
18      "allOf": [+],
19      "css": {+},
20      "description": "A station where entities exits from the system",
21      "name": "Exit"
22    },
23    "Dream.MachineJobShop": {
24      "_class": "node",
25      "allOf": [+],
26      "css": {+},
27      "description": "A station processing items for some time given by a distribution provided by the entities that are
            processed",
28      "name": "Machine"
29    },
30    "Dream.MouldAssembly": {
31      "_class": "node",
32      "allOf": [+],
33      "css": {+},
34      "description": "A station that assembles different components of the same production order into a single entity",
35      "name": "Assembly"
36    },
37    "Dream.MouldAssemblyBuffer": {
38      "_class": "node",
39      "allOf": [+],
40      "css": {+},
41      "description": "A buffer where entities can be hold until the next assembly is ready to process all the entities from the
            same production-order",
42      "name": "Assembly Queue"
43    },
44    "Dream.Operator": {
45      "_class": "node",
46      "allOf": [+],
47      "css": {+},
48      "description": "A resource that operates stations",
49      "name": "Operator"
50    },
51    "Dream.OrderDecomposition": {
52      "_class": "node",
53      "allOf": [+],
54      "css": {+},
55      "description": "A station where production-orders are decomposed into components ",
56      "name": "Decomposition"
57    },
58    "definitions": {
59      "_capacity": {+},
60      "_dist": {+},
61      "_failureDist": {+},
62      "_operationType": {+},
63      "_schedulingRule": {+},
64      "distributionTypes": {
65        "_cauchy": {+},
66        "_exp": {+}, ,
67        "_fixed": {+},
68        "_normal": {+}, ...
69      }
70    }
71  },
```

The DES objects required to model the process of CASE3 pilot case are listed in Listing 5.6. The requirements for the DES object selection is detailed further

in section 5.3. All the DES objects are described in terms of JSON schema nomenclature, inheriting from the generic node object definition. The list of CASE3 instance DES objects contains:

(i) **Dream.OrderDecomposition**: Order decomposition is a buffer like object which receive movable entities (MEs) of type order, and creates all the child-components of that order.

(ii) **Dream.ConditionalBuffer**: Conditional buffers are all the buffers used for modelling the job-shop of CASE3. As explained in section 5.1.3.3 on page 100, certain tasks in order to be performed require that other tasks are concluded. The conditional buffers ensure that no entity will be released for processing (task) unless all precedent tasks are finished.

(iii) **Dream.MachineJobShop**: A type of machine node that inherits from a generic core Dream.Machine object and can be used within a job-shop model. It can handle precedence, as well as complex entity bounded routing and processing times. The operation type property "operationType" (automatic-manual) is handled in combination with the route-task-defined operation types.

(iv) **Dream.MouldAssemblyBuffer**: a type of buffer object that inherits from the conditional buffer, and extends its operation. It collects components that belong to the same parent order, and after collecting them all, it disposes them to a mould assembly object.

(v) **Dream.MouldAssembly**: a machine-job-shop type machine which assembles the components of an order into a mould.

(vi) **Dream.ExitJobShop**: The entity sink of any job-shop model.

The reusable generic definitions that are listed by the *definitions* dictionary contain schemas that describe (i) capacity of buffers, (ii) different types of distributions (processing times, failures), (iii) operation types, and (iv) scheduling rules of buffers or other resources (operators). The use of the definitions is not extended in the CASE3 pilot case, due to the tasks bound processing times, operation types, etc.. There are ideas to extend the use of the *definitions* for validation of data that are not yet described by the JSON embedded schema, as for example the route of a component. For every step of a route, the processing

times would have to comply with the *definitions*' distribution. For more information on the *definitions* dictionary refer to chapter 4 and the fully featured JSON API instance for CASE3 pilot available in Appendix D.

5.3 Input

In section 5.1.3.2 and Figure 5.2 on page 99, we saw the different stages of the whole CASE3 process to be modelled. Similar notation is used for the description of the model within the GUI workflow editor, as we already saw in chapter 3. In Figure 5.3, one can notice the graphical depiction of the model created from within the DREAM GUI editor. Remark that certain nodes are connected by edges, while parts of the model are lacking physical connection with other parts of the model. The routes followed are order dependent and are described by the workplan spreadsheet editor (see Figure 5.4).

After an order arrives, the process commences. The order information should be entered to the order spreadsheet (Figure 5.5), and the workplan spreadsheet (Figure 5.4). The processing of an order starts with CAD design which is performed at an order level. The order is then broken down into different components in a dummy order-decomposition station. After CAD has been completed, the CAM processing of every order component can start; CAM design is performed at component level. For a mould part to start its manufacturing its CAM design should be finished. The different operations (Milling, EDM, Drilling, Turning) on every mould part can be interwoven; the sequence of operations is described in the workplan and decided by the end user (Figure 5.4). These operations are performed on different stations: (i) EDM station, (ii) Milling stations, (iii) Turn station (were Drilling and Turning tasks are performed), and (iv) workbenches for manual operations. In the model, there exist two set of workbenches with identical names, which physically refer to the same workspace, but within the model are used for modelling of different processes. The second set of workbenches is used to model the manual task of assembling the different mould parts. This operation is modelled by a different SE class, namely "Dream.MouldAssembly". Eventually, after all mould parts are manufactured, the mould is assembled. Then the mould is used at the Injection Moulding machine for the production of prototyped parts.

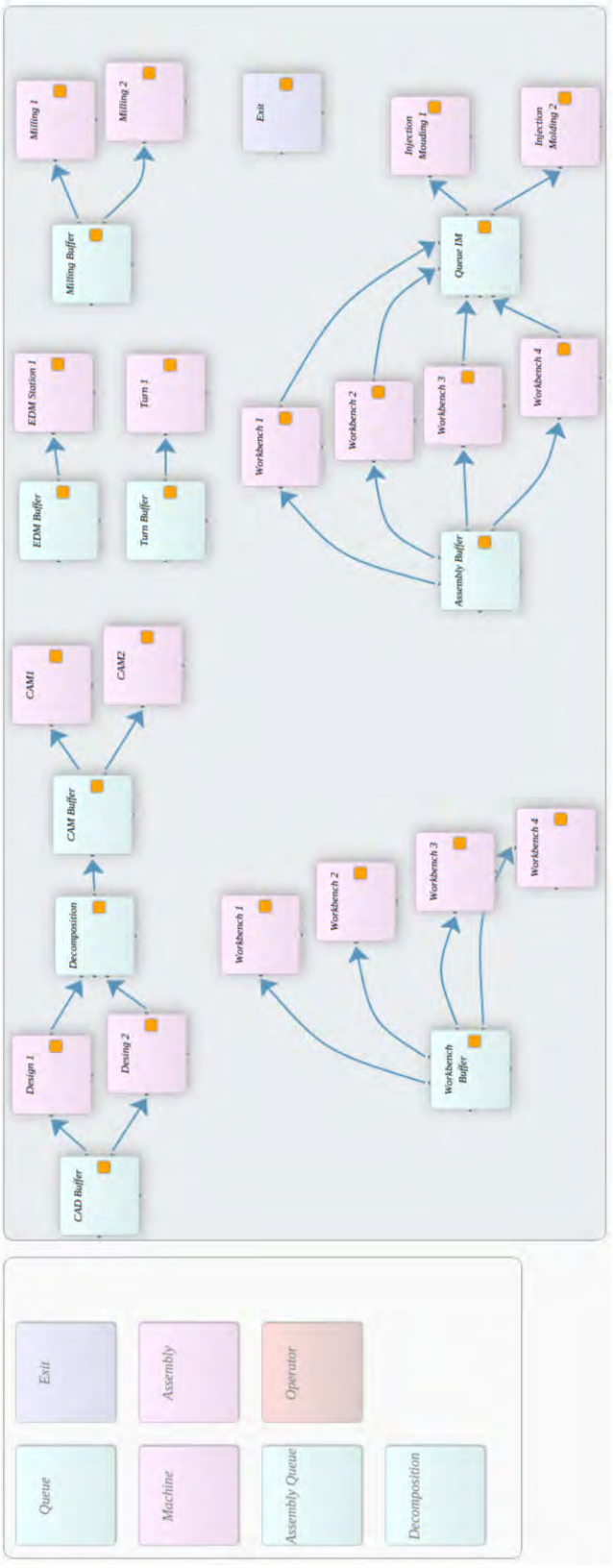
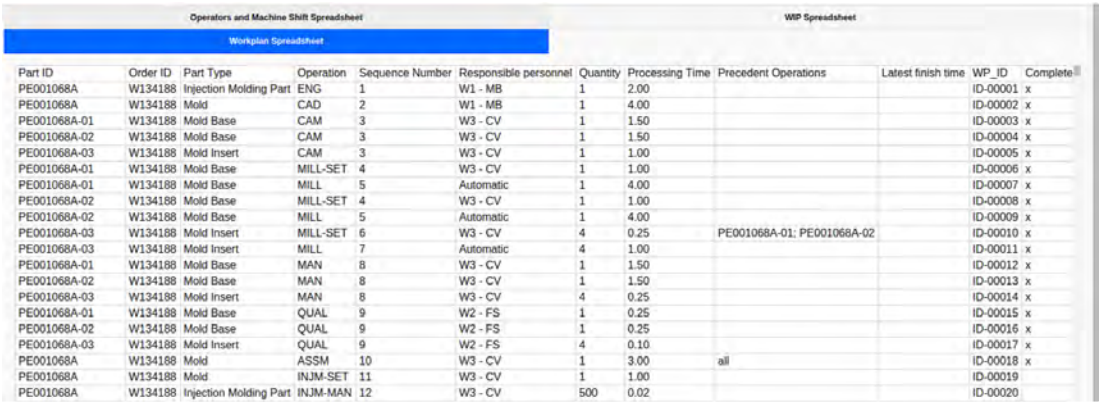


Figure 5.3: CASE3 job-shop model - The model of the CASE3 injection moulding processes



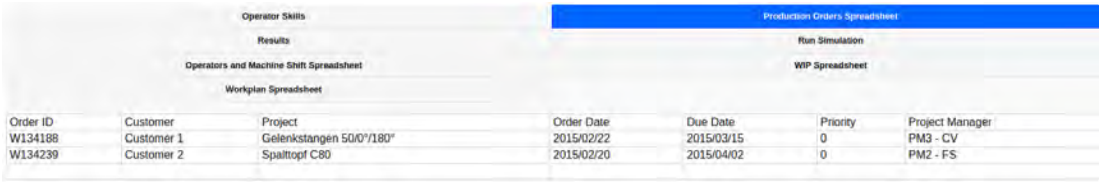
Operators and Machine Shift Spreadsheet | WIP Spreadsheet

Workplan Spreadsheet

Part ID	Order ID	Part Type	Operation	Sequence Number	Responsible personnel	Quantity	Processing Time	Precedent Operations	Latest finish time	WP_ID	Complete
PE001068A	W134188	Injection Molding Part	ENG	1	W1 - MB	1	2.00			ID-00001	x
PE001068A	W134188	Mold	CAD	2	W1 - MB	1	4.00			ID-00002	x
PE001068A-01	W134188	Mold Base	CAM	3	W3 - CV	1	1.50			ID-00003	x
PE001068A-02	W134188	Mold Base	CAM	3	W3 - CV	1	1.50			ID-00004	x
PE001068A-03	W134188	Mold Insert	CAM	3	W3 - CV	1	1.00			ID-00005	x
PE001068A-01	W134188	Mold Base	MILL-SET	4	W3 - CV	1	1.00			ID-00006	x
PE001068A-01	W134188	Mold Base	MILL	5	Automatic	1	4.00			ID-00007	x
PE001068A-02	W134188	Mold Base	MILL-SET	4	W3 - CV	1	1.00			ID-00008	x
PE001068A-02	W134188	Mold Base	MILL	5	Automatic	1	4.00			ID-00009	x
PE001068A-03	W134188	Mold Insert	MILL-SET	6	W3 - CV	4	0.25	PE001068A-01; PE001068A-02		ID-00010	x
PE001068A-03	W134188	Mold Insert	MILL	7	Automatic	4	1.00			ID-00011	x
PE001068A-01	W134188	Mold Base	MAN	8	W3 - CV	1	1.50			ID-00012	x
PE001068A-02	W134188	Mold Base	MAN	8	W3 - CV	1	1.50			ID-00013	x
PE001068A-03	W134188	Mold Insert	MAN	8	W3 - CV	4	0.25			ID-00014	x
PE001068A-01	W134188	Mold Base	QUAL	9	W2 - FS	1	0.25			ID-00015	x
PE001068A-02	W134188	Mold Base	QUAL	9	W2 - FS	1	0.25			ID-00016	x
PE001068A-03	W134188	Mold Insert	QUAL	9	W2 - FS	4	0.10			ID-00017	x
PE001068A	W134188	Mold	ASSM	10	W3 - CV	1	3.00	all		ID-00018	x
PE001068A	W134188	Mold	INJM-SET	11	W3 - CV	1	1.00			ID-00019	
PE001068A	W134188	Injection Molding Part	INJM-MAN	12	W3 - CV	500	0.02			ID-00020	

Figure 5.4: Workplan spreadsheet - The workplan spreadsheet editor of the CASE3 instance

Additional information that cannot be introduced from within the GUI workflow editor are entered manually from the different spreadsheets. Such information can refer to precedence constrains for every task, routing information for every order component, priorities, processing times on every task, order and due dates, machine and operation shifts, and work in progress at the start of the simulation run. As explained already in section 5.2.1 on page 102, there are different spreadsheets for the introduction of this data (Figure 5.6); (i) Production order spreadsheet, (ii) Workplan spreadsheet, (iii) WIP spreadsheet, (iv) Operator and Machine Shift spreadsheet, and (v) Operator skills.



Operator Skills | Production Orders Spreadsheet

Results | Run Simulation

Operators and Machine Shift Spreadsheet | WIP Spreadsheet

Workplan Spreadsheet

Order ID	Customer	Project	Order Date	Due Date	Priority	Project Manager
W134188	Customer 1	Gelenkstangen 50/0°/180°	2015/02/22	2015/03/15	0	PM3 - CV
W134239	Customer 2	Spalttopf C80	2015/02/20	2015/04/02	0	PM2 - FS

Figure 5.5: Production Orders - The Production Orders spreadsheet editor of the CASE3 instance

5.4 Parsing

After the input data are collected, they need to be formatted so that the simulation engine can parse them. The process is performed by pre-processing plugins.

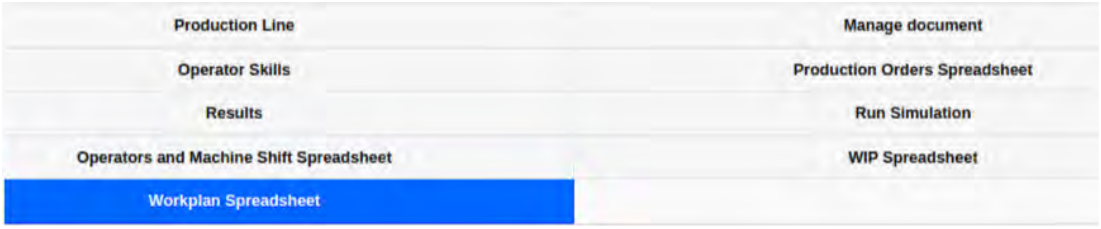


Figure 5.6: CASE3 input views - The input views of the CASE3 instance

All the pre-processing plugins inherit from a generic *InputPreparationPlugin* class (Listing 5.7). The *InputPreparationPlugin* plugin has several methods with the *preprocess* method shared by every pre-processing plugin and is implemented in each child class. This method performs the preprocessing and analysis of the raw input data. Extra methods are available, as for example the *addEdge* method that adds an edge between two given nodes. This method can be handy in the occasion where a node (buffer) is omitted in the GUI designed model. Consider the occasion that the end user requires that the models are designed solely by the use of stations and the buffers are inserted by the platform automatically.

Listing 5.7: Generic *Plugin* and *InputPreparationPlugin*

```
class Plugin(object):
  # Base class for pre-post processing Plugin.
  def __init__(self, logger, configuration_dict):
    self.logger = logger
    self.configuration_dict = configuration_dict
  def getPredecessors(self, data, node_id):
    """returns the predecessors of a node"""
    predecessors=[]
    from copy import copy
    edges=copy(data['graph']['edge'])
    for edge_id,edge in edges.iteritems():
        if edge['destination']==node_id:
            predecessors.append(edge['source'])
    return predecessors
  def getSuccessors(self, data, node_id):
    """returns the successors of a node"""
    successors=[]
    from copy import copy
    edges=copy(data['graph']['edge'])
    for edge_id,edge in edges.iteritems():
        if edge['source']==node_id:
            successors.append(edge['destination'])
    return successors
  def getConfidenceInterval(self, value_list, confidenceLevel):
    """calculate the confidence interval for a list and a confidence level"""
    from dream.KnowledgeExtraction.ConfidenceIntervals import ConfidenceIntervals
    from dream.KnowledgeExtraction.StatisticalMeasures import StatisticalMeasures
    BSM=StatisticalMeasures()
    lb,ub = ConfidenceIntervals().ConfidIntervals(value_list, confidenceLevel)
    return {'lb': lb,
            'ub': ub,
            'avg': BSM.mean(value_list)}
```

```
33    def getAverage(self, value_list):
34      """return the average of a list"""
35      return sum(value_list) / float(len(value_list))
36    def getStDev(self, value_list):
37      """return the standard deviation of a list"""
38      return numpy.std(value_list)
39    def getNameFromId(self, data, node_id):
40      """returns name of a node given its id"""
41      return data['graph']['node'][node_id]['name']
42
43  class InputPreparationPlugin(Plugin):
44    # the generic pre-processing plugin
45    def preprocess(self, data):
46      """Preprocess the data before simulation run."""
47      return data
48    def addEdge(self, data, source, destination, nodeData={}):
49      """adds an edge with the given source and destination"""
50      data['graph']['edge'][source+'_to_'+destination]={
51          "source": source,
52          "destination": destination,
53          "data": {},
54          "_class": "Dream.Edge"}
55      return data
```

The *InputPreparationPlugin* inherits from a generic *plugin* class. All plugins (*ExecutionPlugin*, *OutputPreparationPlugin*, etc.) inherit from that plugin (see Figure 5.7). The methods offered by that plugin can be used or modified by every plugin. They are helpful for designing models (*getPredecessors*, *getSuccessors* methods), or for numerical/statistical analysis and evaluation (*getConfidenceInterval*, *getAverage* etc.). In order to understand better how the manipulation of the raw data is achieved, we will examine the example of the *ReadJSShifts* plugin (Listing 5.3 on page 106).

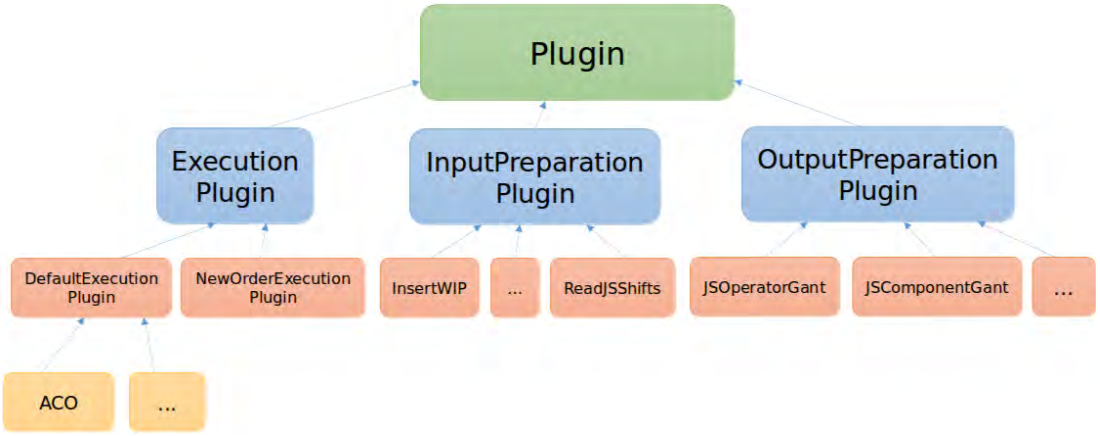


Figure 5.7: Plugins class diagram - The class diagram of the basic plugins

As discussed in section 5.2.1 on page 102, in order to achieve reliable decision support, it is crucial shift schedules are captured by the DES model logic. A

default shift pattern is considered and implemented by the *ReadJSShifts* plugin. A default schedule is hard-coded and reproduced by the logic of the plugin. The end-users are given the chance to define exceptions (in case of absenteeism or a machine break down). The relevant information is introduced to the platform by use of the *Operators and Machines Shift spreadsheet* view; shift start, stop, breaks, and the date are inserted by the end user.

In Listing 5.8, we can see how the *ReadJSShifts* plugin is structured. The code snippet provided in the Listing is simplified. The fully extended plugin is available in Appendix C. The *ReadJSShifts* plugin extends the *preprocess* method of the *InputPreparationPlugin*. It reads the raw shift exception data selected by the end user through the GUI ("output_id": "shift_spreadsheet"), and creates a dictionary of exceptions. Then, it collates the exceptions dictionary with a default shift dictionary, which is produced by the plugin-logic depending on the date that the simulation run starts. For that purpose, the plugin inherits some extra methods from a time calculation plugin named *TimeSupportMixin*. That plugin offers methods that transform date and time into DES simulation engine time units, and vice versa. The *TimeSupportMixin* plugin can be found in Appendix C. After processing the information received (*shiftData*), the plugin returns the formatted information to the next plugin in the plugin list (*return data*).

Listing 5.8: *ReadJSShifts* plugin

```
from dream.plugins import plugin
from dream.plugins.TimeSupport import TimeSupportMixin

class ReadJSShifts(plugin.InputPreparationPlugin, TimeSupportMixin):
  # Input preparation reads the shifts from a spreadsheet
  def preprocess(self, data):
    # reads the shifts from a spreadsheet and updates the interruptions of the corresponding node objects
    self.initializeTimeSupport(data)
    """ input_id : shift_spreadsheet """
    shiftData = data['input'].get(self.configuration_dict["input_id"], [])
    # ======== CREATING THE EXCEPTION SHIFT PATTERN ========
    """iteration through the raw data to structure it into ManPy config"""
    for line in shiftData:
      # if no shift start was given, assume standard 8:00
      startTime = line[2]
      if startTime == '' or startTime == None:
        startTime = "08:00"
      # if no shift end was given, assume standard 18:00
      endTime = line[3]
      if endTime == '' or endTime == None:
        endTime = "18:00"
      """ convert to simulation engine time units"""
      shiftStart = self.convertToSimulationTime(strptime("%s %s" % (line[1], startTime), '%Y/%m/%d %H:%M'))
      shiftEnd = self.convertToSimulationTime(strptime("%s %s" % (line[1], endTime), '%Y/%m/%d %H:%M'))
      """place holders for the shift start and end time """
      timeStartList.append(shiftStart)
```

```
27        timeEndList.append(shiftEnd)
28        """read the off-shift periods"""
29        offshifts = line[-1].replace(" ", "").split(";")
30        for offshift in offshifts:
31          breakStart = self.convertToSimulationTime(strptime("%s %s" % (line[1], offshifts[0]), '%Y/%m/%d %H:%M'))
32          breakEnd = self.convertToSimulationTime(strptime("%s %s" % (line[1], offshifts[1]), '%Y/%m/%d %H:%M'))
33          """place holders for the shift start and end time """
34          timeStartList.append(breakEnd)
35          timeEndList.insert(0, breakStart)
36        """read the entity id"""
37        entityID = line[0].split("-")[0]
38        """update the exceptionShiftPattern dictionary"""
39        for index, start in enumerate(timeStartList):
40          end = timeEndList[index]
41          exceptionShiftPattern[entityID].append([start, end])
42      # ======= CREATE THE DEFAULT SHIFT PATTERN =======
43      # create default pattern for all operators (20 days long)
44      for dayNumber in range(0,20):
45        """check the day, if it is weekend do not create shift entry"""
46        dayDate=now+datetime.timedelta(days=dayNumber)
47        if dayDate.weekday() in [5,6]:
48          continue
49        # -- start and end time of the dayshift --
50        startTime = "08:00"
51        endTime = "18:00"
52        """ convert to simulation engine time units"""
53        shiftStart = self.convertToSimulationTime(strptime("%s %s" % (dayNumber, startTime), '%Y-%m-%d %H:%M'))
54        shiftEnd = self.convertToSimulationTime(strptime("%s %s" % (dayNumber, endTime), '%Y-%m-%d %H:%M'))
55        timeStartList.append(shiftStart)
56        timeEndList.append(shiftEnd)
57      """ for every resource that may have shifts (machines operators)"""
58      for node, node_data in nodes.iteritems():
59        for index, start in enumerate(timeStartList):
60          end = timeEndList[index]
61          defaultShiftPattern[node].append([start, end])
62      # ======== POPULATE THE DEFAULT PATTERN WITH THE EXCEPTIONS ========
63      """ for all the nodes/resources that may have shifts """
64      for node, node_data in nodes.items():
65        """ if the node has exception shift pattern"""
66        if node in exceptionShiftPattern:
67          """loop the exceptions"""
68          for index1, exception in enumerate(exceptionShiftPattern[node]):
69            exceptionDay = math.floor(exception[-1]/24) + 1
70            for index2, default in enumerate(defaultShiftPattern[node]):
71              defaultDay = math.floor(default[-1]/24) + 1
72              """if there is an exception for that day"""
73              if exceptionDay == defaultDay:
74                """ if there is exception update the defaultShiftPattern of the node (operator or machine)"""
75                defaultShiftPattern[node].append(exception)
76                break
77        """update the interruptions of the nodes that have a shifts"""
78        if node in defaultShiftPattern:
79          node_data["interruptions"]["shift"] = {"shiftPattern": defaultShiftPattern.pop(node),"endUnfinished": 0}
80      return data
```

The raw shift data are located under the *input* object of the data model in tabular format. The keyword of the array containing the data is "shift_spreadsheet". The structure of the array after being collected by the GUI spreadsheet widget is presented in Listing 5.9.

Listing 5.9: *input* shift data array

```
"input": {
  "shift_spreadsheet": [[
      "Operator/Machine",
      "Date",
      "Shift Start",
      "Shift Stop",
      "Of-shift periods"
    ],[
      "W1-MB",
      "2015/02/23",
      "09:00",
      "17:00",
      "12:00-14:00"
    ],[
      null,
      "2015/02/24",
      "09:00",
      "17:00",
      ""
    ],...
}
```

As explained in chapter 4, the preprocessing plugins read the raw data, analyse them, update the JSON model accordingly. In the case of the *ReadJSShifts* plugin, the resource-nodes that are subject to interruptions are updated. Under every DES model node (*graph*) that must be interrupted according to the shift spreadsheet information, an extra "interruptions" object is created. This property describes the simulation times when an event should be generated interrupting or re-enabling the corresponding resource-node. In Listing 5.10, we can see an example of an updated node (worker with ID "W7") containing shift intervals information in the form an array. The node, contains extra information as for example the capacity, and the skill-set of the worker ("skillDict").

Listing 5.10: Model's graph-node interruption structure

```
"graph": {
  "node" {
    "W7": {
      "capacity": 1,
```

```
      "name": "W7-WK",
      "skillDict": {
        "load": {+},
        "process": {+},
        "setup": {+}
      },
      "interruptions": {
        "shift": {
          "endUnfinished": 0,
          "shiftPattern": [
            [0, 4.0],
            [18.0, 28.0]
          ]
        }
      },
      "_class": "Dream.Operator",
      "ouputSchedule": 1
    },...
  }
}
```

5.5 Results evaluation

In chapter 4, we saw that after a simulation run (or multiple runs), the simulation engine returns a list of results in the form of a "result_list" array. The results returned by the SE in the CASE3 case are formatted likewise. An example of the returned results can be found in Appendix D.

After the JSON API file is populated with the results returned by the SE, these data need to be structured in such a way that the GUI output widgets-view can read the data, and present them according to the needs of the end user. The task is performed by post-processing plugins. Similar to pre-processing plugins, the post-processing plugins inherit from the same generic plugin (Figure 5.7 and Listing 5.7 on page 115). The basic post-processing plugin *OutputPreparationPlugin* is presented Listing 5.11. The method *postprocess* that performs the analysis and formatting, is not implemented in that plugin; it just returns the data it receives. Specific solutions are implemented explicitly in each inheriting plugin class.

Listing 5.11: *OutputPreparationPlugin* plugin

```
1 class OutputPreparationPlugin(Plugin):
2   def postprocess(self, data):
3     # Postprocess the data after simulation run.
4     return data
```

As example, we will examine the plugin which collects the information relevant to the utilization of the operator resources, and formats the collected data so that they can be presented by the Operator Utilization GUI widget (view_operator_utilization). In Figure 5.8, the corresponding GUI view is presented. In that view, the end-user can evaluate the utilization of every operator.

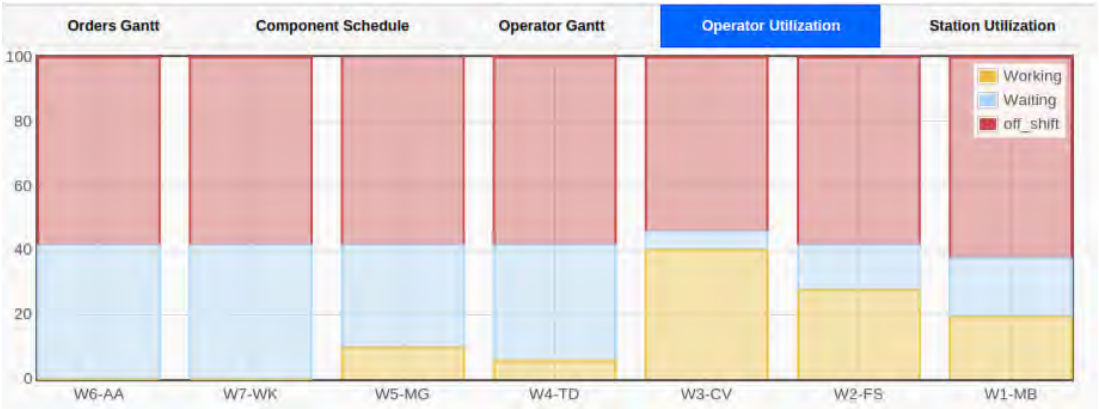


Figure 5.8: Operator Utilization View - The output GUI view that allows the end user evaluate the operators utilization

The plugin which performs the post-processing is named *JSOperatorUtilization*. In Listing 5.12, the corresponding plugin class is shown. Remark that the plugin inherits from the *OutputPreparationPlugin*. The *preprocess* method of the base class is overridden. It reads the "result_list" of the *result* object searches within the "elementList" for operator elements/nodes ("_class" equal to "Dream.Operator").

Listing 5.12: *JSOperatorUtilization* plugin

```
class JSOperatorUtilization(plugin.OutputPreparationPlugin):
  # Output the station utilization metrics in a format compatible with operator utilization widget
  """set of the resource classes"""
  JS_OPERATOR_CLASS_SET = set(["Dream.Operator"])
  def postprocess(self, data):
    for result in data['result']['result_list']:
      ticks = []#placeholder for the element labels on the chart
      working_data = []#holder for working ratios
      waiting_data = []#holder for waiting ratios
      off_shift_data = []#holder for off-shift ratios
      # configuration options of the view (type of chart, etc.)
      options = {
        "xaxis": {
          "minTickSize": 1,
          "ticks": ticks },
        "yaxis": {+},
        "series": {+}
      }
      # holder for the different ratio data
      series = [{
        "label": "Working",
        "data": working_data
      },{
        "label": "Waiting",
        "data": waiting_data
      },{
        "label": "off_shift",
        "data": off_shift_data
      }];
      # the formatted data (dict) is passed to the result object under the id operators_shift
      result[self.configuration_dict['output_id']] = {
        "series": series,
        "options": options
      }

      for i, obj in enumerate(result['elementList']):
        if obj.get('_class') in self.JS_OPERATOR_CLASS_SET:
          objResults=copy(obj['results'])
          if objResults['working_ratio']:
            working_data.append((i, objResults['working_ratio'][0]))
          if objResults['waiting_ratio']:
            waiting_data.append((i, objResults['waiting_ratio'][0]))
          if objResults['off_shift_ratio']:
            off_shift_data.append((i, objResults['off_shift_ratio'][0]))
          """updating the labels for each element with index i"""
          ticks.append((i, obj.get('name', self.getNameFromId(data, obj['id']))))
    return data
```

After the *JSOperatorUtilization* plugin extracts the different ratios ("working_ratio", "waiting_ratio", and "off_shift_ratio") of the operator elements, it appends them to an object named "operator_utilization" under every result of the "result_list". A simplified instance of that object, where only two elements are considered ("W7-WK" and "W6-AA"), can be seen in Listing 5.13. Comparing to Figure 5.8, only the first two bars of the bar-chart can be reproduced from Listing 5.13. The keyword "operator_utilization" is defined in the *application_configuration* of the JSON API for the *JSOperatorUtilization* plugin by the property "output_id" (see Listing 5.5). For more details, check the fully featured

JSON API data model file in Appendix D.

Listing 5.13: The structured operator utilization data

```
"result_list": [
  {
    "_class": "Dream.Simulation",
    "score": 0,
    "order_lateness": {+},
    "operator_utilization": {
      "options": {
        "series": {+},
        "xaxis": {
          "minTickSize": 1,
          "ticks": [
            [0,"W7-WK"],
            [1,"W6-AA"]
          ]
        },
        "yaxis": {"max": 100}
      },
      "series": [{
          "data": [
            [0,0],
            [1,0]
          ],
          "label": "Working"
        }, {
          "data": [
            [0,44],
            [1,44]
          ],
          "label": "Waiting"
        }, {
          "data": [
            [0,56],
            [1,56]
          ],
          "label": "off_shift"
      }]
    },
  },...
]
```

The whole range of the post-processing plugins are structured in a similar way. They read the data as they are returned from the SE, they analyse them in order to extract valuable for the end-user information that can be visualized, and structure them so that they conform with various GUI widget requirements.

5.6 Conclusions

The JSON API data model that has been introduced in this work, has been validated against different case studies. A description of every pilot case that was used for the validation of the data model is provided. An in depth description of the DES platform instance configuration for the CASE3 case study is presented.

Several aspects of the DES information flow have been detailed, with the focus drawn on the information analysis and structuring by the different plugins.

The use case scenario presented, provides a proof of the expandability concept of the platform. The structuring of the different plugins, their object oriented configuration, together with the neutrality and expandability of the data model is tested by the job-shop case study requirements. Furthermore, by comparison of the different pilot cases requirements, it is understandable that the DES data model proposed can handle varying DES modelling needs.

The JSON API data model, can be used for:

- the configuration and synchronization of a DES platform according to the needs of the end-users,
- the structuring of DES data according to a neutral, expandable, and flexible DES data representation model,
- validation of DES data and DES models against a DES data schema, which can also be used for the configuration of the platform, and
- exchange of DES data between various systems, modules, and data sources.

It has also been proved, that the JSON API data model can be extended. The embedded schema can incorporate sub-schema for the *input* object of the data model, as regards the structuring of the "BOM" and "WIP" object that are extensively used by the CASE3 instance JSON API file.

6

Conclusions and Future Directions

6.1 Summary

As mentioned the introductory chapter, the current work is mainly focused on the DES data exchange between manufacturing and simulation applications. The scope was to achieve interoperability between different modules of a simulation platform. The solution proposed is an open-source approach. The DES API introduced is easily configurable and expandable; as a consequence the platform can be easily modified, synchronized, and maintained.

The research was conducted as part of the DREAM project. THe aim of the DREAM project was to develop a modular OS DES platform. The developed DES API enables the different modules, Graphical User Interface (GUI), Simulation Engine (SE which is ManPy) and a Knowledge Extraction (KE) to integrate in a expandable manner. This is achieved while the different modules are independent.

The approach that was used rendered possible a DES model description similar to a work flow representation used by COTS solutions available in the market. Similar to commercial tools, the DES models that are designed by the implemented platform, can be described as a network of nodes and edges. This is the most common design process that the DES modellers and process engineers are familiar with.

6.2 Conclusions

The problem of data sharing between manufacturing and simulation applications, internal or external to the DES platform had to be addressed. The diverse modules needed to be interfaced through the use of the DES API. There was a generic need for an interface that could guarantee the interoperability between the different modules of the platform. Complying with the design requirements of the platform, an open-source solution was developed.

Further requirements maintained that the developed interface would be expandable and reconfigurable, ensuring that the DES platform is modifiable, and that the various modules are in position to be synchronized just by configuring the API. A work flow representation of the manufacturing processes were used for the DES model design tool. That also guided the design of the API in such a direction.

After a literature review of the available solutions, and considering the technologies used for the development of the platform, JSON was selected as the data exchange format of the API. The selected approach is innovative as JSON is used for the first time in developing a data model for exchanging DES data. JSON provides the same interoperability as XML within the scope of Discrete Event Simulation. Among other virtues, JSON supports schemas and referencing; this is crucial for validation of the DES data, and for the synchronization of the platform.

Evaluating the available standards and interfaces for manufacturing data exchange, the structure of the data exchange model was decided. In order to overcome the complexity of the existing solutions, and the limitations of the concurrent development of the different DES platform modules, it was decided to structure a new interface.

The interface framework in the form of a JSON data model, or in other words, the JSON based DES API data model was established; it describes the guidelines that the data exchange between various manufacturing and DES modules should follow. Via the data model, the format of the input models created by any GUI DES model editor is defined, as well as the format of the DES results to be presented to the end-user or the modeller. The representation that was

selected for the manufacturing processes can be described by the directed graph nomenclature.

A JSON schema is embedded in the DES data exchange model. The schema is used for the definition of the input requirements of DES entities or processes (buffers, stations, resources, moving entities, etc.). In addition, the JSON embedded schema can be used for the creation of valid DES data, as well as for the validation of the models, addressing in this way the requirement for data consistency.

JSON also offers the option of referencing. Using referencing, the data model caters for the interfacing of different elements of the platform. In this way, the synchronization of the different modules is achieved. Referencing combined with the expandability of the API data model guarantees the interoperability between the existing modules and any future module.

The introduced JSON DES API data model can describe (i) general attributes of the DES experiment, (ii) the topology of the DES model (directed graph), (iii) extra information about movable entities and relations among them (bill of materials), (iv) initial conditions (work in progress), and (v) returned DES results. The proposed layout can be used with different DES simulators and platforms using appropriate parsers.

Different real use-case scenarios has been used for the validation of the DES API data model. By juxtaposition of the different pilot case requirements, it is understandable that the data model proposed can handle various DES modelling needs.

6.3 Contributions and Future Work

In this work, a neutral DES API data model for DES data exchange is proposed; JSON is used within this context for the first time. The JSON API data model that the current research introduces, can be used for:

- the configuration and synchronization of a DES platform following the requirements of the end-users,

- the formatting of DES data according to a neutral, expandable, and flexible DES data representation model (directed graph),
- validation of DES models against an embedded DES data schema,
- the configuration of the GUI workflow editor of the DES platform using embedded DES schema, and
- exchange of DES data between various systems, modules, and data sources.

By means of the different use case scenarios implemented, it has been proven that the JSON API data model can be extended. The embedded schema can incorporate other sub-schemas regarding the structuring of Bill Of Materials and Work In Progress information. A meta-schema describing the whole DES API data model could also be suggested as future work.

An implementation of the DES API data model on a different DES platform, remains to be seen. That would be used as a proof of concept for the neutrality of the data model. Extra work can be done on integrating different external applications to the existing platform via the proposed DES API data model; an ERP or an MES system could be interfaced with the platform providing extra proof for the neutrality and expandability of the data model.

6.3.1 Expansion considerations

A nice place to start any future work would be a schema for the whole DES API data exchange model. A complete external schema for any possible SE class (fixed-moving entities, time and schedules, etc.) would also be an interesting addition. A review of the DES object classes provided and used by different DES tools (commercial and open-source), should help shape a pool of generic JSON objects representing them. Work on interfacing with different SEs may also be suggested for expanding the API.

Further work on different modelling scenarios and pilot cases would also be important in order to extend the API and render it more generic. For example, a calendar (time keeping) structure can be made generic for use in the input-graph definition of a DES model as well as in the results; interruptions definition can also be improved supporting the configuration of multiple interruptions for

a resource. Furthermore, simplifications could be applied to production orders restrictions and constraints (Bill Of Materials object).

The use of plugins is crucial for interfacing different applications with the API. The generic set of plugins for interfacing different modules with the API can also be expanded and generalized. Different use case scenarios are needed, as well as examination of DES case studies from the literature. The set of plugins can be developed into a set of generic parsers focused on DES, and extended for interfacing with modules written in different programming languages.

Appendix A

Lightweight Markup Languages

Table 1: Lightweight Markup Languages

Language	
AsciiDoc	http://www.methods.co.nz/asciidoc/
BBCode	http://www.bbcode.org/reference.php
Creole	http://www.wikicreole.org/wiki/WikiCreolePressRelease
Crossmark	http://www.wikicreole.org/wiki/Crossmark
Deplate	http://deplate.sourceforge.net/
Epytext	http://deplate.sourceforge.net/]
EtText	http://ettext.taint.org/doc/
Haml	http://haml.info/
MakeDoc	http://www.behaimits.com/makedoc/introduction/
Markdown	https://help.github.com/articles/markdown-basics/
POD	http://juerd.nl/site.plp/perlpodtut
Setext	http://docutils.sourceforge.net/mirror/setext/setext_concepts_Aug92.etx.txt
SiSu	http://www.jus.uio.no/sisu/
Texy!	http://texy.info/
Textile	http://txstyle.org/
txt2tags	http://txt2tags.org/
Wikitext	https://help.wikispaces.com/Wikitext

Appendix B

General Properties of pilot case CASE3

- **confidenceLevel** Confidence level for statistical analysis of stochastic experiments.
- **currentDate** The day the experiment starts, in YYYY/MM/DD format.
- **maxSimTime** Length of the simulation run; how many SE time units should the simulation run.
- **numberOfGenerations** The number of Generations; the optimization routine uses the Ant Colony Optimization approach. Every optimizations step can be characterized as generation of ants. Each ant is generated using a particular set of scheduling rules. The ants scoring the highest are used for the production of the next generation.
- **numberOfAntsPerGenerations** Number of ants per generation.
- **numberOfReplications** Number of replications to run. The SE can run multiple replications to increase stochasticity.
- **numberOfSolutions** Number of solutions to be presented to the end-user. The solutions selected are those with the highest scores.
- **processTimeout** Number of seconds before the calculation process is interrupted.
- **seed** When using the same seed, the random number generator produces the same sequence of numbers.

- **timeUnit** The time unit of the SE; used for input and reporting widgets.
- **trace** Create an excel trace file (Yes or No).

Appendix C

Plugins

All the plugins developed within the scope of DREAM can be found online at `https://github.com/nexedi/dream`.

Processing Plugins

Ant Colony Optimization Plugins

The default ant colony optimization plugin; ACO.py.

```
from dream.plugins import plugin
from pprint import pformat
from copy import copy, deepcopy
import json
import time
import random
import operator
import xmlrpclib
import signal
from multiprocessing import Pool

from dream.simulation.Queue import Queue
from dream.simulation.Operator import Operator
from dream.simulation.Globals import getClassFromName

# run an ant in a subrocess. Can be parrallelized.
def runAntInSubProcess(ant):
  ant['result'] = plugin.ExecutionPlugin.runOneScenario(ant['input'])['result']
  return ant

class ACO(plugin.ExecutionPlugin):
  def _calculateAntScore(self, ant):
    """Calculate the score of this ant. Implemented in the Subclass, raises NotImplementedError
    """
    raise NotImplementedError("ACO subclass must define '_calculateAntScore' method")

  def createCollatedScenarios(self,data):
    """creates the collated scenarios, i.e. the list of options collated into a dictionary for ease of referencing in ManPy
    to be implemented in the subclass
    """
    raise NotImplementedError("ACO subclass must define 'createCollatedScenarios' method")

  # creates the ant scenario based on what ACO randomly selected
```

```
34    def createAntData(self,data,ant):
35      """creates the ant scenario based on what ACO randomly selected.
36      raises NotImplementedError
37      """
38      raise NotImplementedError("ACO subclass must define 'createAntData' method")
39
40
41    def run(self, data):
42      """Preprocess the data.
43      """
44      distributor_url = data['general'].get('distributorURL')
45      distributor = None
46      if distributor_url:
47          distributor = xmlrpclib.Server(distributor_url)
48
49      multiprocessorCount = data['general'].get('multiprocessorCount')
50
51      tested_ants = set()
52      start = time.time() # start counting execution time
53
54      collated=self.createCollatedScenarios(data)
55      assert collated
56
57      max_results = int(data['general'].get('numberOfSolutions',0))
58      assert max_results >= 1
59
60      ants = []#list of ants for keeping track of their performance
61
62      # Number of times new ants are to be created, i.e. number of generations (a
63      # generation can have more than 1 ant)
64      seedPlus = 0
65      for i in range(int(data["general"]["numberOfGenerations"])):
66          scenario_list = []# for the distributor
67          # number of ants created per generation
68          for j in range(int(data["general"]["numberOfAntsPerGenerations"])):
69              # an ant dictionary to contain rule to queue assignment information
70              ant = {}
71              # for each of the machines, rules are randomly picked from the
72              # options list
73              seed = data['general'].get('seed', 10)
74              if seed == '' or seed == ' ' or seed == None:
75                  seed = 10
76              for k in collated.keys():
77                  random.seed(seed+seedPlus)
78                  ant[k] = random.choice(collated[k])
79                  seedPlus +=1
80              # TODO: function to calculate ant id. Store ant id in ant dict
81              ant_key = repr(ant)
82              # if the ant was not already tested, only then test it
83              if ant_key not in tested_ants:
84                  tested_ants.add(ant_key)
85                  ant_data=deepcopy(self.createAntData(data, ant))
86                  ant['key'] = ant_key
87                  ant['input'] = ant_data
88                  scenario_list.append(ant)
89
90          if distributor is None:
91              if multiprocessorCount:
92                  self.logger.info("running multiprocessing ACO with %s processes" % multiprocessorCount)
93                  # We unset our signal handler to print traceback at the end
94                  # otherwise logs are confusing.
95                  sigterm_handler = signal.getsignal(signal.SIGTERM)
96                  pool = Pool(processes=multiprocessorCount)
97                  try:
98                      signal.signal(signal.SIGTERM, signal.SIG_DFL)
99                      scenario_list = pool.map(runAntInSubProcess, scenario_list)
100                     pool.close()
101                     pool.join()
102                 finally:
103                     signal.signal(signal.SIGTERM, sigterm_handler)
```

```
104             else:
105                 # synchronous
106                 for ant in scenario_list:
107                     ant['result'] = self.runOneScenario(ant['input'])['result']
108 
109         else: # asynchronous
110             self.logger.info("Registering a job for %s scenarios" % len(scenario_list))
111             start_register = time.time()
112             job_id = distributor.requestSimulationRun(
113                 [json.dumps(x).encode('zlib').encode('base64') for x in scenario_list])
114             self.logger.info("Job registered as %s (took %0.2fs)" % (job_id, time.time() - start_register ))
115 
116             while True:
117                 time.sleep(1.)
118                 result_list = distributor.getJobResult(job_id)
119                 # The distributor returns None when calculation is still ongoing,
120                 # or the list of result in the same order.
121                 if result_list is not None:
122                     self.logger.info("Job %s terminated" % job_id)
123                     break
124 
125             for ant, result in zip(scenario_list, result_list):
126                 result = json.loads(result)
127                 if 'result' in result: # XXX is this still needed ???
128                   result = result['result']
129                   assert "result_list" in result
130                 else:
131                   result = {'result_list': [result]}
132                 ant['result'] = result
133 
134         for ant in scenario_list:
135             ant['score'] = self._calculateAntScore(ant)
136 
137         ants.extend(scenario_list)
138 
139         # remove ants that outputs the same schedules
140         # XXX we in fact remove ants that produce the same output json
141         ants_without_duplicates = dict()
142         for ant in ants:
143             ant_result, = copy(ant['result']['result_list'])
144             ant_result['general'].pop('totalExecutionTime', None)
145             ant_result = json.dumps(ant_result, sort_keys=True)
146             ants_without_duplicates[ant_result] = ant
147 
148         # The ants in this generation are ranked based on their scores and the
149         # best (max_results) are selected
150         ants = sorted(ants_without_duplicates.values(),
151           key=operator.itemgetter('score'))[:max_results]
152 
153         for l in ants:
154             # update the options list to ensure that good performing queue-rule
155             # combinations have increased representation and good chance of
156             # being selected in the next generation
157             for m in collated.keys():
158                 # e.g. if using EDD gave good performance for Q1, then another
159                 # 'EDD' is added to Q1 so there is a higher chance that it is
160                 # selected by the next ants.
161                 collated[m].append(l[m])
162 
163     data['result']['result_list'] = result_list = []
164     for ant in ants:
165       result, = ant['result']['result_list']
166       result['score'] = ant['score']
167       result['key'] = ant['key']
168       result_list.append(result)
169 
170     self.logger.info("ACO finished, execution time %0.2fs" % (time.time() - start))
171     return data
```

Appendix

The modified ant colony optimization plugin for the CASE3 pilot case is JobShopACO.py.

```
from pprint import pformat
from copy import copy, deepcopy
import time

from dream.simulation.Queue import Queue
from dream.simulation.Operator import Operator
from dream.simulation.Globals import getClassFromName
from dream.plugins.ACO import ACO

class JobShopACO(ACO):
  def _calculateAntScore(self, ant):
    """Calculate the score of this ant.
    """
    totalDelay=0 #set the total delay to 0
    result, = ant['result']['result_list'] #read the result as JSON
    #loop through the elements
    throughput=0
    for element in result['elementList']:
        # find the throughput of the solution
        element_family = element.get('family', None)
        if element_family=='Exit':
            results=element['results']
            throughput=results.get('throughput', "0")[0]

        #id the class is Job
        if element_family == 'Job':
            results=element['results']
            delay = float(results.get('delay', "0"))
            # A negative delay would mean we are ahead of schedule. This
            # should not be considered better than being on time.
            totalDelay += max(delay, 0)
    production_orders_spreadsheet=ant['input']['input'].get('production_orders_spreadsheet',[])
    # find the number of orders
    numberOfOrders=0
    for row in production_orders_spreadsheet:
        if row[0] and (not row[0]=="Order Id"):
            numberOfOrders+=1

    # if in this solution not all orders finished give the total delay as infinite so that it want be considered positive
    if not throughput==numberOfOrders:
        totalDelay=float('inf')

    return totalDelay

  # creates the collated scenarios, i.e. the list
  # of options collated into a dictionary for ease of referencing in ManPy
  def createCollatedScenarios(self,data):
    collated = dict()
    for node_id, node in data['graph']['node'].items():
      node_class = getClassFromName(node['_class'])
      if issubclass(node_class, Queue) or issubclass(node_class, Operator):
        collated[node_id] = list(node_class.getSupportedSchedulingRules())
    return collated

  # creates the ant scenario based on what ACO randomly selected
  def createAntData(self,data,ant):
    # set scheduling rule on queues based on ant data
    ant_data = copy(data)
    for k, v in ant.items():
        ant_data["graph"]["node"][k]['schedulingRule'] = v
    return ant_data
```

Pre-processing Plugins

The pre-processing plugin that collects the shift (exceptions) information, and creates the shift patterns for the different resources; ReadJSShifts.py.

```
from copy import copy
import json
import math
import time
import random
import copy
import datetime
from operator import itemgetter

from dream.plugins import plugin
from dream.plugins.TimeSupport import TimeSupportMixin

class ReadJSShifts(plugin.InputPreparationPlugin, TimeSupportMixin):
  """ Input preparation
      reads the shifts from a spreadsheet
  """

  def correctTimePair(self, start, end):
    '''takes a pair of times and returns the corrected pair according to the current time'''
    # if the end of shift shift already finished we do not need to consider in simulation
    if start<0 and end<=0:
      return None
    # if the start of the shift is before now, set the start to 0
    if start<0:
      start=0
    # sometimes the date may change (e.g. shift from 23:00 to 01:00).
    # these would be declared in the date of the start so add a date (self.timeUnitPerDay) to the end
    if end<start:
      end+=self.timeUnitPerDay
    return (start, end)

  def preprocess(self, data):
    """ reads the shifts from a spreadsheet and updates the interruptions of the corresponding node objects
    """
    strptime = datetime.datetime.strptime
    # read the current date and define dateFormat from it
    try:
      now = strptime(data['general']['currentDate'], '%Y/%m/%d %H:%M')
      # calculate the hours to end the first day
      hoursToEndFirstDay = datetime.datetime.combine(now.date(), datetime.time(23,59,59)) - datetime.datetime.combine(
            now.date(), now.time())
      data['general']['dateFormat']='%Y/%m/%d %H:%M'
    except ValueError:
      now = strptime(data['general']['currentDate'], '%Y/%m/%d')
      hoursToEndFirstDay = datetime.time(23,59,59)
      data['general']['dateFormat']='%Y/%m/%d'
    self.initializeTimeSupport(data)

    shiftData = data["input"].get("shift_spreadsheet",[])
    nodes = data["graph"]["node"]

    defaultShiftPattern = {}#default shift pattern dictionary (if no pattern is defined for certain dates)
    exceptionShiftPattern = {}# exceptions for shift pattern dictionary as defined in the spreadsheet

    if shiftData:
      shiftData.pop(0)
      #iteration through the raw data to structure it into ManPy config
      lastrec=None
      for line in shiftData:
        # if all the records of that line are none then continue
        toContinue = False
        for record in line:
          if record != None and record!='':
```

```
63                toContinue = True
64                break
65            if not toContinue:
66              continue
67            # list to hold the working intervals start times
68            timeStartList = []
69            # list to hold the working intervals end times
70            timeEndList = []
71            #if no shift start was given, assume standard 8:00
72            startTime = line[2]
73            if startTime == '' or startTime == None:
74              startTime = "08:00"
75            shiftStart = self.convertToSimulationTime(strptime("%s %s" % (line[1], startTime), '%Y/%m/%d %H:%M'))
76            #if no shift end was given, assume standard 18:00
77            endTime = line[3]
78            if endTime == '' or endTime == None:
79              endTime = "18:00"
80            shiftEnd = self.convertToSimulationTime(strptime("%s %s" % (line[1], endTime), '%Y/%m/%d %H:%M'))
81            timePair = self.correctTimePair(shiftStart, shiftEnd)
82            if not timePair:
83              continue
84            else:
85              shiftStart, shiftEnd = timePair
86              timeStartList.append(shiftStart)
87              timeEndList.append(shiftEnd)
88
89            if line[-1]:
90              offshifts = line[-1].replace(" ", "").split(";")
91              for offshift in offshifts:
92                limits = offshift.split("-")
93                breakStart = self.convertToSimulationTime(strptime("%s %s" % (line[1], limits[0]), '%Y/%m/%d %H:%M'))
94                breakEnd = self.convertToSimulationTime(strptime("%s %s" % (line[1], limits[1]), '%Y/%m/%d %H:%M'))
95                timePair = self.correctTimePair(breakStart, breakEnd)
96                if not timePair:
97                  continue
98                else:
99                  breakStart, breakEnd = timePair
100                 timeStartList.append(breakEnd)
101                 timeEndList.insert(0, breakStart)
102           # sort the list before proceeding
103           timeEndList.sort()
104           timeStartList.sort()
105           #if it is the current row is an extended row for the information belonging to a resource, and no resource name is
                  entered
106           if line[0]:
107             entityID = line[0].split("-")[0]
108           else:
109             entityID = ""
110           if str(entityID) == '':
111             #take it as a continuation for the last entered resource
112             for index, start in enumerate(timeStartList):
113               end = timeEndList[index]
114               if not start and not end:
115                 continue
116               exceptionShiftPattern[lastrec].append([start, end])
117           #if resource name is defined
118           elif str(entityID) not in exceptionShiftPattern:
119             #take the name of the last entered resource from here
120             lastrec = str(entityID)
121             exceptionShiftPattern[lastrec] = []
122             for index, start in enumerate(timeStartList):
123               end = timeEndList[index]
124               if not start and not end:
125                 continue
126               # if there is no other entry
127               if not len(exceptionShiftPattern[lastrec]):
128                 exceptionShiftPattern[lastrec] = [[start, end]]
129               else:
130                 exceptionShiftPattern[lastrec].append([start, end])
131           #to avoid overwriting existing records, if there is another entry for a resource but does not follow it immediately (e.g.
```

```
            W2-FS)
        else:
          lastrec = str(entityID)
          #extend the previous entry for the resource
          for index, start in enumerate(timeStartList):
            end = timeEndList[index]
            if not start and not end:
              continue
            exceptionShiftPattern[lastrec].append([start, end])

      #sorts the list in case the records were not entered in correct ascending order
      for info in exceptionShiftPattern:
        exceptionShiftPattern[info].sort(key=itemgetter(0))

      #create default pattern for all operators (10 days long)
      timeStartList = []
      timeEndList = []

      for dayNumber in range(0,20):
        # check the day, if it is weekend do not create shift entry
        dayDate=now+datetime.timedelta(days=dayNumber)
        if dayDate.weekday() in [5,6]:
            continue
        startTime = "08:00"
        endTime = "18:00"
        upDate = now.date()+datetime.timedelta(days=dayNumber)
        shiftStart = self.convertToSimulationTime(strptime("%s %s" % (upDate, startTime), '%Y-%m-%d %H:%M'))
        shiftEnd = self.convertToSimulationTime(strptime("%s %s" % (upDate, endTime), '%Y-%m-%d %H:%M'))
        timePair = self.correctTimePair(shiftStart, shiftEnd)
        shiftStart, shiftEnd = timePair
        timeStartList.append(shiftStart)
        timeEndList.append(shiftEnd)
      #for every operator (can be also machine) create an entry on the defaultShiftPattern
      for node, node_data in nodes.iteritems():
        #if the node is an operator
        if node_data.get('_class', None) == 'Dream.Operator':
          for index, start in enumerate(timeStartList):
            end = timeEndList[index]
            if not start and not end:
              continue
            if not node in defaultShiftPattern:
              defaultShiftPattern[node] = [[start, end]]
            else:
              defaultShiftPattern[node].append([start, end])

      for node, node_data in nodes.items():
        if node_data.get('_class', None) == 'Dream.Operator':
          modifiedDefaultDays = []# the days of the defaultShiftPattern that have been modified according to the
              exceptionShiftPattern
          if node in exceptionShiftPattern:
            for index1, exception in enumerate(exceptionShiftPattern[node]):
              # XXX think of the case where the exception starts one day and finishes the next
              # calculate the time difference in hours from the end of the first day to the end of the exception
              # check if we are still in the first day
              if hoursToEndFirstDay.total_seconds()/3600 > exception[-1]:
                exceptionDay = 0
              # calculate the number of days till the end of the exception
              else:
                exceptionDay = math.floor((exception[-1] - hoursToEndFirstDay.total_seconds()/3600)/24) + 1
              # check the weekday
              exceptionDate=now+datetime.timedelta(days=exceptionDay)
              # if it is weekend create shift entry for that date. The default pattern does not need to be changed
              if exceptionDate.weekday() in [5,6]:
                  defaultShiftPattern[node].append(exception)
              # for exceptions in weekdays
              else:
                  for index2, default in enumerate(defaultShiftPattern[node]):
                    # check if we still are in the first day
                    if hoursToEndFirstDay.total_seconds()/3600 > default[-1]:
                      defaultDay = 0
```

```
200                     # calculate the number of days till the end of the default shift
201                     else:
202                       defaultDay = math.floor((default[-1] - hoursToEndFirstDay.total_seconds()/3600)/24) + 1
203                     if exceptionDay == defaultDay:
204                       # update the defaultShiftPattern of the node (operator or machine)
205                       # if the exception day has not been modified then delete the previous entry and use the first exception
                              that occurs
206                       if not exceptionDay in modifiedDefaultDays:
207                         defaultShiftPattern[node][index2] = exception
208                       # otherwise append it at the end
209                       else:
210                         defaultShiftPattern[node].append(exception)
211                       modifiedDefaultDays.append(exceptionDay) # the day has been modified, add to the modified days
212                       break
213           # update the interruptions of the nodes that have a defaultShiftPattern
214           if node in defaultShiftPattern:
215             # sort the shift pattern of every node
216             defaultShiftPattern[node].sort(key=itemgetter(0))
217             # get the interruptions of the object
218             interruptions = node_data.get("interruptions", {})
219             if not interruptions:
220               node_data["interruptions"] = {}
221             node_data["interruptions"]["shift"] = {"shiftPattern": defaultShiftPattern.pop(node),
222                                                    "endUnfinished": 0}
223
224     return data
```

Time-support Plugin

The plugin for time conversion support - TimeSupportMixin.py.

```
1  from datetime import datetime
2  from datetime import timedelta
3  import random
4  from pprint import pformat
5
6  from dream.plugins import plugin
7
8  class TimeSupportMixin(object):
9    """Helper methods to handle time units.
10   """
11   initialized = False
12
13   def initializeTimeSupport(self, data):
14     self.timeUnit = data['general']['timeUnit']
15     if self.timeUnit == 'second':
16       self.timeUnitPerDay = 24* 60* 60.
17     elif self.timeUnit == 'minute':
18       self.timeUnitPerDay = 24* 60.
19     elif self.timeUnit == 'hour':
20       self.timeUnitPerDay = 24.
21     elif self.timeUnit == 'day':
22       self.timeUnitPerDay = 1.
23     elif self.timeUnit == 'week':
24       self.timeUnitPerDay = 1/ 7.
25     elif self.timeUnit == 'month':
26       self.timeUnitPerDay = 1/ 30.
27     elif self.timeUnit == 'year':
28       self.timeUnitPerDay = 1/ 360.
29     else:
30       raise ValueError("Unsupported time unit %s" % self.timeUnit)
31
32     self.dateFormat = data['general'].get('dateFormat', '%Y/%m/%d %H:%M:%S')
33     # Convert simulation 0 time to real world time
34     self.now = datetime.strptime(data['general']['currentDate'], self.dateFormat)
```

```
35      self.initialized = True
36
37    def convertToRealWorldTime(self, simulation_time):
38      """Convert simulation clock time to real world time, as python datetime object.
39      """
40      assert self.initialized, "initializeTimeSupport has not been called"
41      assert not simulation_time==float("inf"), "cannot convert infinite simulation time to real world time"
42      return self.now + timedelta(days=simulation_time/self.timeUnitPerDay)
43
44    def convertToFormattedRealWorldTime(self, simulation_time):
45      """Convert simulation clock time to real world time, as string formatted according to general dateFormat
46      """
47      assert self.initialized, "initializeTimeSupport has not been called"
48      return (self.now + timedelta(days=simulation_time/self.timeUnitPerDay)).strftime(self.dateFormat)
49
50    def convertToTimeStamp(self, simulation_time):
51      """Convert simulation clock time to real world time, as timestamp.
52      """
53      return (self.convertToRealWorldTime(simulation_time) - datetime(1970, 1,1)).total_seconds()
54
55    def convertToSimulationTime(self, real_world_time):
56      """Convert real world time (as python datetime object) to simulation clock time.
57      """
58      assert self.initialized, "initializeTimeSupport has not been called"
59      # get the seconds difference and divide by (24*60*60/self.timeUnitPerDay)
60      # 24*60*60 is timeUnitPerDay if the unit is seconds
61      return (real_world_time - self.now).total_seconds()/(86400/self.timeUnitPerDay)
62
63    def getTimeUnitText(self):
64      """Return the time unit as text.
65      """
66      assert self.initialized, "initializeTimeSupport has not been called"
67      return self.timeUnit
```

Appendix D

JSON DES API instance for pilot case CASE3

Due to the length of the appendix, it is omitted from this printed copy. The reader can obtain a full version of the document at the following links:

- `https://www.dropbox.com/s/fc3ygv7kf4d6tw7/CASE3.json?dl=0`
- `http://tinyurl.com/oks9hqd`
- `http://www.jsoneditoronline.org/?id=52677a9bb45bb0982220176bd6dea92b`
- `http://tinyurl.com/q2mth54`

The web-pages `http://www.jsoneditoronline.org/` and `http://jsonviewer.stack.hu/` can be used for large JSON files viewing, as the user can fold and unfold the JSON objects for grouped viewing.

References

Apple Inc. (2010), *Property List Programming Guid* [online], available: https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/PropertyLists/PropertyLists.pdf [accessed: 13 May 2015]. 23

Axelsson, J., Birbeck, M., Dubinko, M., Epperson, B., Ishikawa, M., McCarron, S., Navarro, A. and Peberton, S. (2010), *XHTML 2.0* [online], available: http://www.w3.org/TR/xhtml2/ [accessed: 13 May 2015]. 15

Bacon, J., Tardella, N., Pratt, J., Hu, J. and English, J. (2006), 'The Surgical Simulation and Training Based Language for Medical Simulation', *Medicine Meets Virtual Reality 14: Accelerating Change in Healthcare: Next Medical Toolkit* **119**, 37–42. **URL:** *http://doi.acm.org/10.1145/155360.155368#-#* 26

Banks, J., Carson, J., Nelson, B. and Nicol, D. (2010), *Discrete-event system simulation 5th ed.*, Prentice Hall. 2

BenKiki, O., Evans, C. and dot Net, I. (2009), *YAML Ain't Markup Language (YAML^{TM} Version 1.2* [online], available: http://www.yaml.org/spec/1.2/spec.html [accessed: 13 May 2015]. 18

Bergmann, S., Stelzer, S. and Straßburger, S. (2011), 'Initialization of Simulation Models Using CMSD', *In Proceedings of the Winter Simulation Conference*, 2011, Phoenix, Arizona, 2228-2239. 4, 36

Birkenheuer, G., Blunk, D., Breuers, S., Brinkmann, A., dos Santos Vieira, I., Fels, G., Gesing, S., Grunzke, R., Herres-Pawlis, S., Kohlbacher, O. et al. (2012), 'MoSGrid: efficient data management and a standardized data exchange format for molecular simulations in a grid environment', *Journal of Cheminformatics* **4**(1), 1–1. 26

Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauß, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., Neumerkel, D., Olsson, h., J. V., P. and

Wolf, S. (2011), 'The functional mockup interface for tool independent exchange of simulation models', *Proceedings of the 8th International Modelica Conference, Dresden*, 2011, 20 - 22. 26

Bloomfield, R., Mazhari, E., Hawkins, J. and Son, Y. (2012), 'Interoperability of manufacturing applications using the Core Manufacturing Simulation Data (CMSD) standard information model', *Computers and Industrial Engineering* **62**(4), 1065 – 1079. 32

Borgo, S. and Leitao, P. (2004), 'The role of foundational ontologies in manufacturing domain applications', *In Proceedings of On The Move to Meaningful Internet Systems 2004 CoopIS, DOA and ODBASE, pt 1*, 2004, 670-688. 4

Boulonne, A., Johansson, B., Skoogh, A. and Aufenanger, M. (2010), 'Simulation data architecture for sustainable development', *In Proceedings of the 2010 Winter Simulation Conference*, December, 2010, 3435-3446. 5, 27

Bray, T., Paoli, J., Sperberg–McQueen, C., Maler, E., Yergeau, F. and Cowan, J. (2006), *Extensible Markup Language (XML) 1.1 (Second Edition)* [online], available: http://www.w3.org/TR/xml11/ [accessed: 13 May 2015]. 14

Burnett, G., Medeiros, D., Finke, D. and Traband, M. (2008), 'Automating the development of shipyard manufacturing models', *In Proceedings of the 2008 Winter Simulation Conference*, December, 2008, 1761-1767. 6

Carey, C. (2008), *Protocol Buffers - Googles's data interhcange format* [online], available: https://github.com/google/protobuf2009 [accessed: 13 May 2015]. 21

Carson II, J. (2005), 'Introduction to modeling and simulation', *In Proceedings of the Winter Simulation Conference*, December, 2005, 16-23. 5

Catron, B. and Ray, S. (1991), 'ALPS: A language for process specification', *International Journal of Computer Integrated Manufacturing* **4**(2), 105–113. 3

Chartrand, G. (2006), *Introduction to graph theory*, Tata McGraw-Hill Education. 49

Chasseur, C., Li, Y. and Patel, J. (2013), 'Enabling JSON Document Stores in Relational Systems.', *Proceedings 16th International Workshop on the Web and Databases (WebDB)*,2013,1-6. 18

Core Manufacturing Simulation Data Product Development Group (2010), 'Standard for: Core manufacturing simulation data uml model'. 31

Crockford, D. (2006), *The application/json Media Type for Javascript Object Notation (JSON)* [online], available: http://tools.ietf.org/html/rfc4627 [accessed: 13 May 2015]. 17, 58

Dagkakis, G., Heavey, C., Robin, S. and Perrin, J. (2013), 'ManPy: An Open-Source Layer of DES Manufacturing Objects Implemented in SimPy', *Proceedings of 8th EUROSIM Congress on Modelling and Simulation (EUROSIM)*, 2013, 357-363. 41

DeCuetos, P., Seyrat, C. and Thienot, C. (2006), *Binary MPEG format for XML (BiM)* [online], available: http://mpeg.chiariglione.org/standards/mpeg-b/binary-mpeg-format-xml [accessed: 13 May 2015]. 17

Dimyadi, J., Spearpoint, M. and Amor, R. (2008), 'Sharing Building Information using the IFC Data Model for FDS Fire Simulation', *Proceedings of the ninth internation symposium of Fire Safety Science*, 2008, 1329-1340. 25

Drath, R. (2010), *Datenaustausch in der Anlagenplanung mit AutomationML: Integration von CAEX, PLCopen XML und COLLADA*, Springer-Verlag. 31

Drath, R., Luder, A., Peschke, J. and Hundt, L. (2008), 'AutomationML-the glue for seamless automation engineering', *Proceedings of IEEE International Conference on Emerging Technologies and Factory Automation ETFA 2008*, 2008, 616-623. 30

ECMA International (2013), *Standard ECMA–404: The JSON Data Interchange Format* [online], available: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf [accessed: 13 May 2015]. 17

Fagin, R., Kolaitis, P., Miller, R. and Popa, L. (2005), 'Data exchange: semantics and query answering.', *Theoretical Computer Science* **336**(1), 89–124.
URL: *http://www.biomedcentral.com/1752-0509/5/198* 12

Falkman, P., Nielsen, J., Lennartson, B. and von Euler-Chelpin, A. (2008), 'Generation of step ap214 models from discrete event systems for process planning and control', *IEEE Transactions on Automation Science and Engineering* **5**(1), 113–126. 30

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. (1999), *Hypertext Transfer Protocol – HTTP/1.1*, Technical report, Network Working Group. 41

REFERENCES

Fowler, J., Rose, O. and Fowler, J. (2004), 'Grand challenges in modeling and simulation of complex manufacturing systems', *Simulation* **80**, 9–469. 3

Furfari, F., Potortì, F. and Savic, D. (2008), *The CostGlue XML schema*, Technical report, Technical report. CNR-ISTI, via Moruzzi 1. Report. 26

Galiegue, F. and Court, G. (2013), *JSON Schema: core definitions and terminology json-schema-core* [online], available: http://json-schema.org/latest/json-schema-core.html [accessed: 13 May 2015]. 18, 49

Gang, G., Min, W., Ge, L. and Gang, X. (2010), 'Architecture and Standard Proposals for Next Generation Modeling and Simulation', in society for modeling & simulation international, *In the Proceedings of the 2010 Conference on Grand Challenges in Modeling & Simulation*, 2012, Ottawa, Canada, 12-19.
URL: *http://dl.acm.org/citation.cfm?id=2020619.2020621* 4, 5

Gautier, L. (2008), *A simple and efficient access to R from Python* [online], available: http://rpy.sourceforge.net/rpy2 [accessed: 11 June 2015]. 42

Gifford, C. (2007), *The Hitchhiker's Guide to Operations Management: ISA-95 Best Practices Book 1.0*, ISA. 31

Gregersen, J., Gijsbers, P. and Westen, S. (2007), 'OpenMI: Open modelling interface', *Journal of Hydroinformatics* **9**(3), 175–191. 26

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., Frystyk Nielsen, H., Karmarkar, A. and Lafon, Y. (2005), *SOAP Version 1.2 Part 1: Messaging Framework (Second Edtion)* [online], available: http://www.w3.org/TR/soap12/ [accessed: 13 May 2015]. 15

Harward, G. and Harrell, C. (2006), 'Assessment of the NIST Shop Data Model as a Neutral File Format', *In the Proceedings of the Winter Simulation Conference, 2006. WSC 06*, December, 2006, 941-946. 5, 28

Heavey, C., Dagkakis, G., Barlas, P., Papagiannopoulos, I., Robin, S., Mariani, M. and Perrin, J. (2014), 'Development of an open-source discrete event simulation cloud enabled platform', *Proceedings of the 2014 Winter Simulation Conference*, 2014, 2824 - 2835. 39

Heilala, J., Montonen, J., Kuula, T., Usenius, T., Maantila, M. and Sillanpaa, J. (2011), 'User-centric development of simulation based manufacturing operation planning and scheduling system', *Proceedings of the 2011 IEEE International Symposium on Assembly and Manufacturing (ISAM)*, 2011, 1-6. 36

Hickson, I., Berjon, R., Faulkner, S., Leithead, T., D. Navara, E., O'Connor, E. and Pfeiffer, S. (2014), *HTML5* [online], available: http://www.w3.org/TR/html5/ [accessed: 11 June 2015]. 40

Hossain, M., Harari, N., Semere, D., Martensson, P., Ng, A. and Andersson, M. (2012), 'Integrated Modeling and Application of Standardized Data Schema', *In Proceedings of 5th Swedish Production Symposium, (SPS12)*, 2012. 5

Hucka, M., Finney, A., Sauro, H., Bolouri, H., Doyle, J., Kitano, H., Arkin, A., Bornstein, B., Bray, D., Cornish-Bowden, A., Cuellar, A., Dronov, S., Gilles, E., Ginkel, M., Gor, V., Goryanin, I., Hedley, W., Hodgman, T., Hofmeyr, J., Hunter, P., Juty, N., Kasberger, J., Kremling, A., Kummer, U., Le Novere, N., Loew, L., Lucio, D., Mendes, P., Minch, E., Mjolsness, E., Nakayama, Y., Nelson, M., Nielsen, P., Sakurada, T., Schaff, J., Shapiro, B., Shimizu, T., Spence, H., Stelling, J., Takahashi, K., Tomita, M., Wagner, J. and Wang, J. (2003), 'The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models', *Bioinformatics* **19**(4), 524 – 531. 25, 26

Information processing systems – Open Systems Interconnection (1987), 'Specification of abstract syntax notation one (asn.1)'. 12

Jackson, E., Murri, D., Hill, M., Jessick, M., Penn, J., Hasan, D., Crues, E., Falck, R., McCarthy, T., Vuong, N. and Zimmerman, C. (2011), 'Assessment of the Draft AIAA S-119 Flight Dynamic Model Exchange Standard', *Proceedings of 2011 AIAA Modelling and Simulation Technologies Conference*, 2011. 26

Jacobs, S. (2005), *Open Node Syntax (ONX)* [online], available: http://www.seairth.com/web/onx/onx.html [accessed: 13 May 2015]. 22

Johansson, M., Johansson, B., Skoogh, A., Leong, S., Riddick, F., Lee, Y., Shao, G. and Klingstam, P. (2007), 'A Test Implementation of the Core Manufacturing Simulation Data Specification', in ieee press, *In Proceedings of the 39th Conference on Winter Simulation*, 2007, Piscataway, USA, 1673-1681.
URL: *http://dl.acm.org/citation.cfm?id=1351542.1351843* 5, 26

Kilgore, R. (2001), 'Web 1: open source simulation modeling language (SML)', *Proceedings of the 33nd conference on Winter simulation*, 2001, 607 - 613. 28

Kjellberg, T., von Euler-Chelpin, A., Hedlind, M., Lundgren, M., Sivard, G. and Chen, D. (2009), 'The machine tool modelA core part of the digital factory', *CIRP Annals-Manufacturing Technology* **58**(1), 425–428. 30

Laasonen, M. (2010), 'Data exchange from BIM to building-use simulation', *Proceedings of the International Conference on Computing in Civil and Building Engineering*, 2010. 25

Lacy, L. (2006), *Interchanging Discrete event simulation Process Interaction Models using the Web Ontology Language-OWL*, PhD thesis, University of Central Florida Orlando, Florida. vii, 4, 32, 33

Law, A. and Kelton, D. (1991), *Simulation Modeling and Analysis*, McGraw-Hill. 2

Lee, J., Kang, H., Do Noh, S., Woo, J. and Lee, P. (2011), 'NESIS: a neutral schema for a web-based simulation model exchange service across heterogeneous simulation software', *International Journal of Computer Integrated Manufacturing* **24**(10), 948–969. 29

Lengyel, E. (2013), *Open Data Description Language (OpenDDL)* [online], available: http://openddl.org/ [accessed: 14 May 2015]. 24

Lengyel, E. (2015), 'Open game engine exchange specification version 1.1.2'. 24

Leong, S., Lee, Y. and Riddick, F. (2006), 'A core manufacturing simulation data information model for manufacturing applications', *In Proceedings of the Simulation Interoperability Workshop, Simulation Interoperability and Standards Organization*, 1-7. 4, 5, 32, 36

Leuck, D. (2012), *Simple Declerative Language (SDL)* [online], available: http://ikayzo.org/display/SDL/Home [accessed: 13 May 2015]. 22

Luder, A., Hundt, L. and Keibel, A. (2010), 'Description of manufacturing processes using AutomationML', *Proceedings of the 2010 IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2010. 30

Manufacturing Enterprise Solutions Association (MESA International) (2013), *Business To Manufacturing Markup Language (B2MML)* [online], available: http://www.mesa.org/en/B2MML.asp [accessed: 14 May 2015]. 31

McLean, C., Lee, Y., Shao, G., Riddick, F. and Leong, S. (2005), *Shop data model and interface specification*, US Department of Commerce, Technology Administration, National Institute of Standards and Technology. 27

McLean, C., Leong, S., Harrell, C., Zimmerman, P. and Lu, R. (2003), 'Simulation standards: current status, needs, future directions, panel: simulation standards: current status, needs, and future directions', *In Proceedings of the 35th conference on Winter simulation: driving innovation*, 2003, 2019-2026. 4, 6, 27

Moorthy, S. (1999), 'Integrating the CAD model with dynamic simulation: simulation data exchange', *Proceedings of 1999 Winter Simulation Conference*, 1999, 276 - 280. 27

Nance, R. (1993), 'A History of Discrete Event Simulation Programming Languages', *SIGPLAN Not.* **28**(3), 149–175.
URL: *http://doi.acm.org/10.1145/155360.155368#-#* 2

Neema, H., Gohl, J. and Lattmann, Z. (2014), 'Model-based integration platform for FMI co-simulation and heterogeneous simulations of cyber-physical systems', *Proceedings of the 10th International Modelica Conference*, 2014. 25

Nisbet, N. and Liebich, T. (2007), 'ifcXML Implementation Guide', *International Alliance for Interoperability* . 26

Nottingham, M. and Sayre, R. (2005), *The Atom Syndication Format* [online], available: http://tools.ietf.org/html/rfc4287 [accessed: 13 May 2015]. 15

Nurseitov, N., Paulson, M. and Reynolds, R. (2009), 'Comparison of JSON and XML Data Interchange Formats: A Case Study', *Caine* pp. 157–162. 35

Nytsch-Geusen, C., Klempin, C., Nunez v. Voigt, J. and Radler, J. (2003), 'Integration of CAD, thermal building simulation and CFD by using the IFC data exchange format', *Proceedings of the eighth Internation IBPSA Conference*, 2003, 967-973. 25

Ochmanska, E. (2004), Computational Science - ICCS 2004, in Bubak, M. and van Albada, G. and Sloot, P. and Dongarra, J., eds., *An Approach to Web-Oriented*

Discrete Event Simulation , Springer Berlin Heidelberg, 527-531.
URL: *http://dx.doi.org/10.1007/978-3-540-24685-5_82* 4

Pease, R. and Carrico, T. (1998), *JTF ATD core plan representation*, Technical report, Technical Report SS-97-06. 3

Pratt, M. (2001), 'Introduction to ISO 10303the STEP standard for product data exchange', *Journal of Computing and Information Science in Engineering* **1**(1), 102–103. 29

PrestonWerner, T. (2015), *Tom's Obvious, Minimal Language* [online], available: https://github.com/toml-lang/toml [accessed: 13 May 2015]. 21

Python Software Foundation (2010), *Python Language Reference, version 2.7* [online], available: https://www.python.org [accessed: 11 June 2015]. 41

Raggett, D., Le Hors, A. and Jacobs, I. (1999), *HTML 4.01 Specification* [online], available: http://www.w3.org/TR/REC-html40/ [accessed: 13 May 2015]. 13

Ramrez, Y. and Nembhard, D. (2004), 'Measuring knowledge worker productivity', *Journal of Intellectual Capital* **5**(4), 602–628.
URL: *http://dx.doi.org/10.1108/14691930410567040* 4

Randell, L. (2002), *On Discrete-Event SImulation and integration in th manufacturing system development process*, PhD thesis, Division of Robotics - Department of Mecahinical Engineering - Lund University. 2

Reichenthal, S. (2002), 'Re-introducing web-based simulation', *Proceedings of 2002 Winter Simulation Conference*, 2002, 847 - 852. 28

Rhodes, J. (2000), *Just-In-Time Information Retrieval JITIR*, PhD thesis, MIT, Cambridge, MA (2000 June). 47

Riddick, F. and Lee, Y. (2008), 'Representing layout information in the CMSD specification', *In Proceedings of the 2008 Winter Simulation Conference*, 2008, 1777–1784. 32

Robinson, S., Nance, R., Paul, R., Pidd, M. and Taylor, S. (2004), 'Simulation model reuse: definitions, benefits and obstacles', *Simulation Modelling Practice and Theory* **12**(78), 479 – 494.
URL: *http://www.sciencedirect.com/science/article/pii/S1569190X04000760* 5

Rohl, M. and Uhrmacher, A. (2005), 'Flexible integration of XML into modeling and simulation systems', *In Proceedings of the Winter Simulation Conference, 2005*, December, 2005, 8 pp.-. 6

RSS Advisory Board (2009), *RSS 2.0 Specification* [online], available: http://www.rssboard.org/rss-specification [accessed: 13 May 2015]. 15

Runde, S. and Fay, A. (2008), 'A Data Exchange Format for the Engineering of Building Automation Systems', *Proceedings of the IEEE International Conference on emerging technologies and factory automation*, 2008, 303-310. 25

Schlenoff, C., Gruninger, M., Tissot, F., Valois, J., Lubell, J. and Lee, J. (2000), *The process specification language (PSL) overview and version 1.0 specification*, Citeseer. 3

Schneider, J., Kamiya, T., Peintner, D. and Kyusakov, R. (2014), *Efficient XML Interchange (EXI) Format 1.0 (Second Edition)* [online], available: http://www.w3.org/TR/exi/ [accessed: 13 May 2015]. 17

Scholten, B. (2007), *The road to integration: A guide to applying the ISA-95 standard in manufacturing*, Isa. 31

Shariatzadeh, N., Sivard, G., Hedlind, M. and Wickman, C. (2012), 'Material flow data representation and integration based on STEP', *In Proceedings of 2012 IEEE Conference on Control, Systems Industrial Informatics (ICCSII)*, September, 2012, 30-35. 3, 5

Sly, D. and Moorthy, S. (2001), 'Simulation data exchange (SDX) implementation and use', *In Proceedings of the Winter Simulation Conference*, 2001, 1473-1477. 5, 27

Smirnov, N. (1944), 'Approximate laws of distribution of random variables from empirical data', *Uspekhi Matematicheskikh Nauk* (10), 179–206.
URL: *http://www.biomedcentral.com/1752-0509/5/198* 46

Srinivasan, R. (1995), *External Data Representation Standard* [online], available: https://www.ietf.org/rfc/rfc1832.txt [accessed: 13 May 2015]. 13

Stark, J. (2011), *Product lifecycle management*, Springer. 29

REFERENCES

Tate, A. (1995), 'Characterizing Plans As a Set of Constraints—the <I-N-OVA> Model—a Framework for Comparative Analysis', *SIGART Bull.* **6**(1), 26–32.
URL: *http://doi.acm.org/10.1145/202187.202193* 4

Team SimPy (2015), *SimPy* [online], available: https://simpy.readthedocs.org/en/latest/ [accessed: 11 June 2015]. 40

Tricker, R. (2014), *ISO 9001: 2008 for Small Businesses 5th edition*, Routledge. 97

Umeda, S. and Jones, A. (1997), 'Simulation in Japan: State-of-the-art update', *Animation* **18**, 5–15. 26

Waltemath, D., Adams, R., Bergmann, F., Hucka, M., Kolpakov, F., Miller, A., Moraru, I., Nickerson, D., Sahle, S., Snoep, J. and Le Novere, N. (2011), 'The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models', *BMC Systems Biology* **5**.
URL: *http://www.biomedcentral.com/1752-0509/5/198* 25, 26

Wiedemann, T. (2002), 'Next generation simulation environments founded on open source software and XML-based standard interfaces', *Proceedings of the 2002 Winter Simulation Conference, Vols 1 and 2*, 2002, 623 - 628. 28