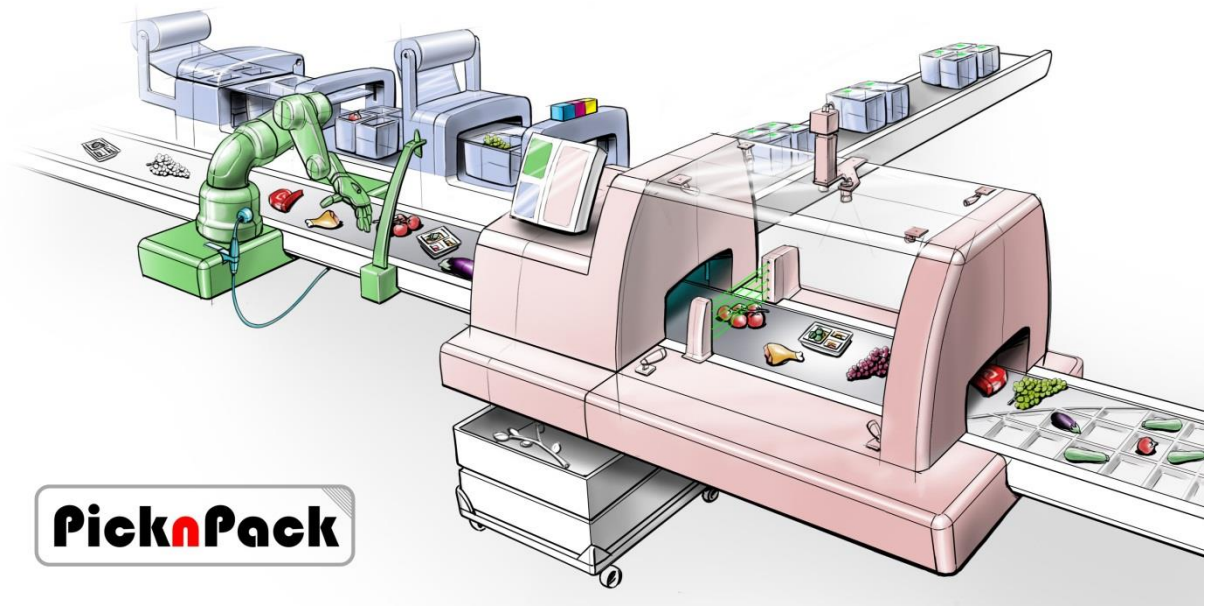
D2.3 - Deliverable accompanying Milestone MS3

Overview of all demonstrated tasks

Johan Philips



Flexible robotic systems for automated adaptive packaging of fresh and processed food products

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n° 311987.

Table of Contents

1 Introduction

This deliverable captures the **set of software design patterns** that were developed and applied during Pick-n-Pack and **how they improved the flexibility and traceability in the food packaging context**. Each pattern is brlefly described and applied to one of the (sub)goals of Pick-n-Pack.

2 Communication mechanisms

One important objective of Pick-n-Pack was to **allow all partners to connect to each other and share data efficiently**. This also included **interfacing to legacy software and hardware** in a flexible and easy manner. In typical distributed systems in the food (packaging) industry, **setting up such connections is cumbersome and requires manual configurations or hardcoded settings**. In Pick-n-Pack, we wanted to make a **step change** by having the systems **auto-discover and auto-configure** themselves.

To address these issues a communication mechanism over **network sockets** was devised as it naturally decouples (software) systems. Nevertheless, to goal was to **adopt the same communication patterns at network level** (e.g. between Pick-n-Pack modules) **as well as at system level** (e.g. between devices on one Pick-n-Pack module). Therefore, we introduced **state of the art solutions from the IT and networking community** with respect to discovery and connection setup and applied it to the food packaging context of the Pick-n-Pack line.

This involved a review of **open source communication middleware** which were able to provide such solutions **at all levels of granularity** (e.g. at network line level and at system module level).

2.1 Pick-n-Pack data bus

The **Pick-n-Pack data bus defines the communication infrastructure** on which messages and events can be exchanged between Modules, the Line Controller and other software entities such as the Tracing Database, the World Model and the Flexible GUI. It makes use of ZeroMQ, an open source network socket library, for **messaging** and Zyre for **local discovery and connection setup** protocols. This allows more **flexibility and dynamism** since software entities (representing for instance a Pick-n-Pack Module) following the Pick-n-Pack data bus protocols can **dynamically join and leave** the Pick-n-Pack data bus and all connections to other entities are configured automatically.

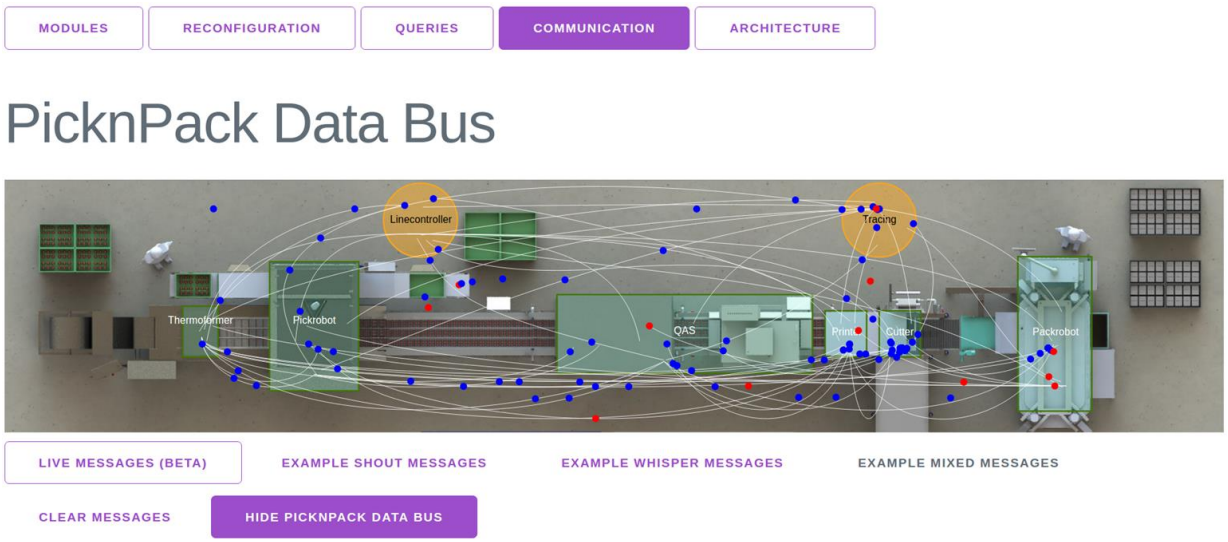


Figure 1: Pick-n-Pack data bus visualised in Flexible GUI showing connections and message exchange between peers.

The Pick-n-Pack data bus allows two messages mechanisms: **whispering** and **shouting**. The former is used for sending a particular message to one particular *peer* on the Pick-n-Pack data bus, while the latter is used for group messaging, i.e. multicasting messages to a set of peers.

Figure 1 shows a visualisation of the Pick-n-Pack data bus in the Flexible GUI, where all the available Modules and the Line Controller are connected and exchanging messages.

2.2 JSON message models

In order to standardise communication messages passing the Pick-n-Pack data bus, a **message model** was designed. Also for sharing data, similar models had to be developed to allow flexible data storage and tracing. In this section we will focus on the message models.

The Pick-n-Pack messages are **represented in JSON** and this format was chosen, since it has a lot of traction in the world wide web community. This also allowed us to efficiently integrate and interpret the message in web applications which were used for visualisation.

Each message consists of a **header** and **payload** to first identify the type of message, its sender and receiver. Additionally, **a URI of the data model and meta model** can be supplied in the message to make it **self-descriptive**. However, in the Pick-n-Pack project this feature was not well adopted yet.

An example of such a message to define a label for a tray of cooked chicken is written below. It is a message from the **world model**, which **maintains a sliding window of all data** gathered on the Pick-n-Pack data bus, with respect to the trays and products on the Pick-n-Pack Line.

The message payload contains all the quality assessment data on that particular tray and product relevant for the label to be printed. Noticable here is also the introduction of universally unique identifiers (UUIDs). These identifers not only facilitate tracing afterwards but are also used to determine the location of each tray on the line and to perform actions accordingly. This also enhances flexibility with respect to which products are currently where.


```
{
 "metamodel": "pnp_msgs",
 "model": "URI",
 "type": "LABEL",
 "sender": "worldmodel",
 "receiver": "printer",
 "payload": {
        "batch_uuid": "7581fbbd-d05d-4d89-9d1d-054d11f4be58",
        "tray_uuid": "6d2fa5e8-c2cb-4d93-ba58-47ee1f9953f4",
        "product_description": "Chicken breasts",
        "weight": "459",
        "packaging_date": "20160429",
        "packer_name": "Pick-n-Pack",
        "packer_address": "Droevendaalsesteeg 4, 6708 PB Wageningen",
        "origin": "The Netherlands",
        "region": "Wageningen",
        "product": "CHICKEN BREAST",
        "variety": "Val Dieu",
        "specs": {
                "class": "I",
                "calories": "145.8kcal",
                "carbohydrate": "0.1g",
                "protein": "24.8g",
                "fat": "4.6g",
                "fibre": "0.2g",
                "expiry_date": "20160505"
        }

 }
}
```


2.3 Queries

Another communication mechanism that was introduced in Pick-n-Pack was that of **stateful queries**. The rationale behind it is to **move away from stateless data exchange** which does not contain **context** and introduce **dialogs**. The bidirectionality of a dialog makes a **trade-off between scalability and reliability** in a sense that you need to be aware of which peer you are sending messages to or receiving messages from but it **allows handshakes** to ensure important data is sent and received. As a technical advantage it also allows

controllable throttle, where event sources can limit their speed upon request, which is not possible in traditional publish – subscribe communication.

In Pick-n-Pack queries were used to communicate with the **ontology** that was defined in the project around tomato trusses. For instance, the Flexible GUI would send a query for **converting particular units** from the metric system to the Imperial system.

Another link was with the **Semantic database** that was used to store data from the Pick-n-Pack data bus. Again the Flexible GUI could query this database to **visualise particular events** currently being sent around in the Pick-n-Pack line or, alternatively, query for past data to solve some **tracing** request.

3 Life-cycle state machine

All Pick-n-Pack Modules and their devices have to **work together to accomplish the food packaging task**. The **Line Controller** was introduced to manage this process. To increase flexibility here with respect to **identifying problems** in one or more modules and **knowing when the line can proceed**, the **Life-Cycle State Machine** was designed.

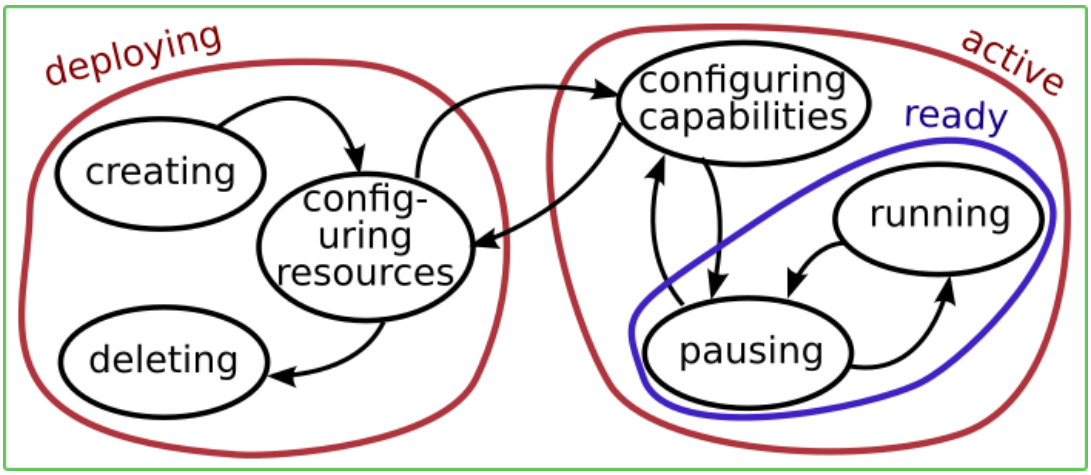


Figure 2: Life Cycle State Machine

This Life-Cycle State Machine defines a **set of states** each Pick-n-Pack Module has to go through before it can safely join the food packaging task. This consists of allocating the right **resources**, setting up **communication**, **configuring** the correct software for the task at hand (e.g. analysing and packaging tomatos or chicken breasts) and **running** its particular task. In Figure 2 a diagram of the Life-Cycle State Machine is depicted.

3.1 Start-up procedure

One objective of the Life Cycle State Machine is to provide a **homogeneous startup procedure** for the wholePick-n-Pack line. This includes software entities at **different levels of abstraction**, such as the Line Controller at line level, individual Modules at module level and Devices within a particular Module as this state machine is applied hierarchically.

At start-up, each module starts in the **deploying** state which consists of **creating** the required data structures, **allocating** adequate memory for these data structures and **configuring its resources**. The latter **explicitly defines the hierarchical connection** between the levels of abstraction. Resources for a Module would be Devices (e.g. a camera for the QAS Module), while resources for a Line would be the set of Modules required for the line to go in to production. It also includes **computational** resources, such as CPU time and **communication** resources such as network sockets or interprocess pipes.

Once this configuring state has completed, the software entity moves into the **active** state, which again is a **composition** of other states. Here the **capabilities** of this particular software entity are first **configured**, which sets policies and configuration parameters of the tasks it is responsible for. For example, the QAS Module would calibrate its camera specifically for assessing the quality of tomatoes and load all the correct algorithms for this task.

Finally, after all configuration is finished, the software entity moves into the **ready** state, where is it able to execute the computations. For example, in the **running** state of the Thermoformer module, a new mould of a particular tray layout is created.

3.2 Stop light protocol

Since all Modules in the Pick-n-Pack line express the same Life Cycle State Machine, this information can be used by the Line Controller to introspect the Modules and **confirm if it is safe to continue operation or not.** The Stop light protocol works bottom up, in such a way that the Modules communicate their state as **ready (green), busy (orange) or not ready (red)** and the Line Controller only sends a **go signal** to all Modules in the Pick-n-Pack line if it received a green light from all Modules. This go results in the next step in the **stop & go procedure** of the line. Each go a new tray is created and the line shifts.

3.3 Reconfiguration

Additionally, adoption of the Life Cycle State Machine facilitates reconfiguration, both in software and hardware. One of the goals of the Pick-n-Pack project was the flexible and efficient reconfiguration of the Line from, for example, packaging tomato trusses to packaging cooked chicken breasts.

This can now be done by emiting a **reconfiguration signal**, either by the Line Controller or directly by an operator from within the Flexible GUI. This signal is transmitted onto the Pick-n-Pack data bus and each Module can move from the ready state back into the configuring capabilities state to reconfigure its parameters and policies to conform to the task request. For some Modules this can be done purely in software, such as the QAS Module, while others require manual intervention, such as the robot arm which would require a different gripper. The **same stop light protocol and Life Cycle State Machine come into play** here. In its configuring capabilities state, the software entity representing this robot arm would inform the

operator that a gripper replacement is required. **Only after this manual action has completed, it would move into the ready state.** Since the Line Controller checks the stop lights on all Modules it will be informed that this particular Module is not ready yet and the **Pick-n-Pack line will effectively halt until all manual intervention is executed**.

4 Flexible GUI

At the Wageningen workshop in May 2016, we demonstrated **Version 1** of the flexible GUI, which collected data from the different Modules in the Pick-n-Pack Line (by listening on the Pick-n-Pack databus). During the Summer of 2016, this flexible GUI was further improved and new functionality was added, which resulted in the demonstration of **Version 2** at the Holbeach workshop in September 2016.

Demonstrations of both Version 1 and Version 2 with recorded data from the Pick-n-Pack Line have been deployed on Heroku and are available at and respectively.

The **architecture** of the web application is depicted in Figure 3. This shows how the flexible GUI, running partly on the Web-app server and partly in the browser, receives data from the Pick-n-Pack databus, which uses the “Zyre network”. This is done using the **Mediator pattern**, which listens to the databus, converts this data to useable data for the web visualisation and adds this data to the **semantic** realtime database.

The user interface itself is built in a **modular** way, meaning that it will visualise active Modules on the Pick-n-Pack databus and **dynamically update** events it receives from them through the Mediator.

For demonstration purposes, the Flexible GUI also includes an interface to **reconfigure** the line. This is shown in Figure 4. The visualisation indicates the states of the modules that need to reconfigured, in this case the QAS, and displays the underlying **Life-Cycle State Machine** that each Module implements to achieve this behaviour.

In addition, a **Query interface** was made availabe, to query the **semantic database** and visualise statistics on the quality assessment or generate labels for the Printer Module. An example is given in Figure 5.

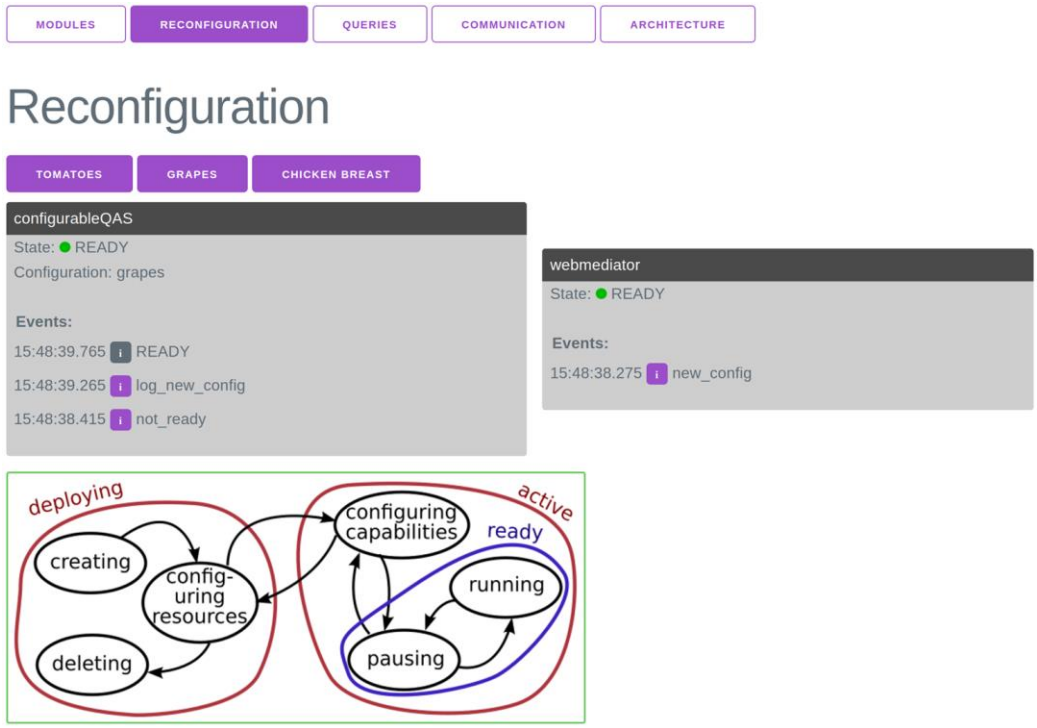


Figure 4: Reconfiguration

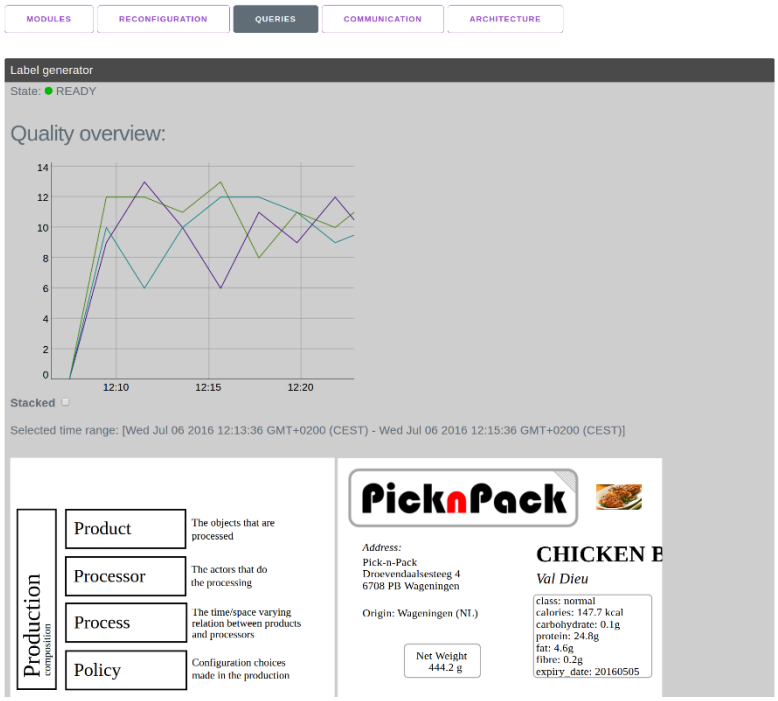


Figure 5: Quality assessment overview

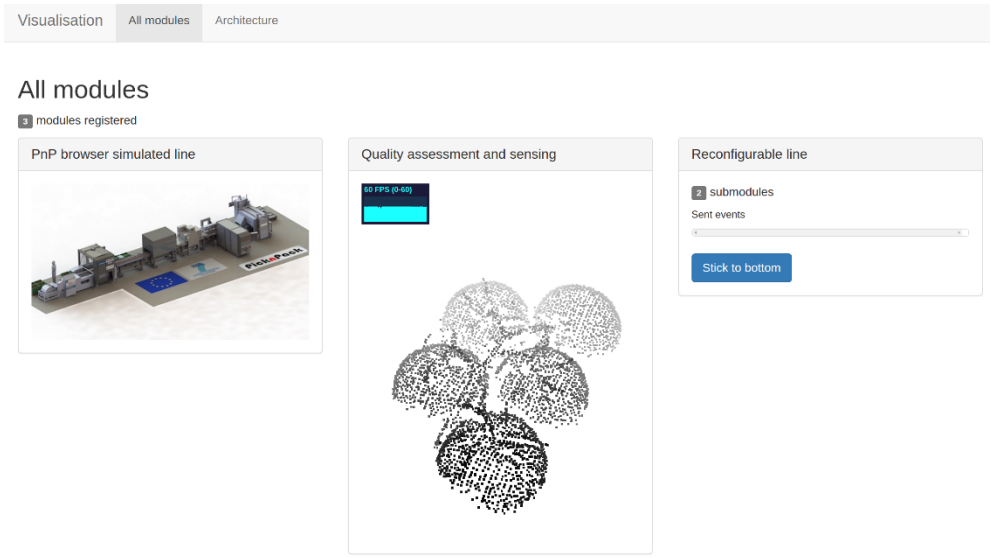


Figure 6: Version 2 of Flexible GUI

In Version 2, the design of the web application was further **modularised** by adding a hierarchy and creating a **generic visualisation framework** which can be reused **independent of Pick-n-Pack**. The same functionality was provided as demonstrated in Version 1, but visualisations where improved and regrouped in an intuitive way. Also an **additional 3D view of analysed tomato trusses** by the QAS Module was developed. Figure 6 demonstrates the modularity of the Flexible GUI Version 2.

5 Mediator

5.1 Decoupling

The Mediator pattern was developed to decouple software entities which share data but not necessarily need to be aware of each others knowledge domain. An example is the visualisation of data collected on the Pick-n-Pack databus and visualised in the Flexible GUI. The **Mediator** has **knowledge** about both the agreed data format on the Pick-n-Pack databus and data format used by the visualisation framework.

Another way of mediation was done at the Module level, where internal specific knowledge of the Module developer needed to be converted to comply with the agreed **Pick-n-Pack language**.

Complementary to the Mediator pattern, the Zyre wrapper, developed by KUL, offers the technological solution to Module developers to share data and send events on the Pick-n-Pack databus.

The adoption by all partners of both the Mediator pattern and the Zyre wrapper, or connection to the Pick-n-Pack databus directly, allowed for faster integration and easier data sharing and contributed to a succesfull demonstration of the Pick-n-Pack Line at Wageningen and Holbeach workshops.

5.2 Hierarchical design: Plant – Line – Module – Device – Algorithm

As was mentioned in previous sections, the same design patterns that were applied on the Pick-n-Pack Line, could be applied at a **higher level of abstraction** such as a Plant with several Lines, or a **finer grained level** such as individual Modules, Devices or even Algorithms. This is not only what makes these patterns flexible as required by the Pick-n-Pack objectives, but also generic such that t**hey can be used outside the context of this project**.

Within the Pick-n-Pack project, the QAS Module clearly showed the benefits of applying these patterns also to its local Devices and algorithms. They reused the same communication mechanisms between systems and within systems and applied to Life Cycle State Machine at both levels of abstraction.

6 Data models for traceability

Another important objective of the Pick-n-Pack project was the design of a generic data model that contributed to flexible and efficient traceability. One design choices that facilitates tracing was the introduction of Universally Unique Identifiers or UUIDs for all different aspects of a productions. This allows to store only a small footprint in the form of meta-data in the tracing database and keeping all other raw data locally at the Module that created this data. Via the UUIDs, it is possible to trace back to which Module performed which tasks on which tray of tomatoes at what moment in time and configured with which policies. This led to the generic model for production, the 5P model.

6.1 5P - generic production model

This model describes what (meta) data we need to store to effectively trace back to a particular **production**. Any **product**, e.g. a package of tomato trusses, involves **processors**, e.g. a particular Module, that executed a set of **processes**, e.g. assessing quality in several ways, with a set of **policies**, e.g. configuration parameters of calibrations of the cameras. Within the Pick-n-Pack project the adoption of this 5P model was not complete, but initial steps were made in this direction. Figure 7 depicts the relation and function of each of the 5 P's.

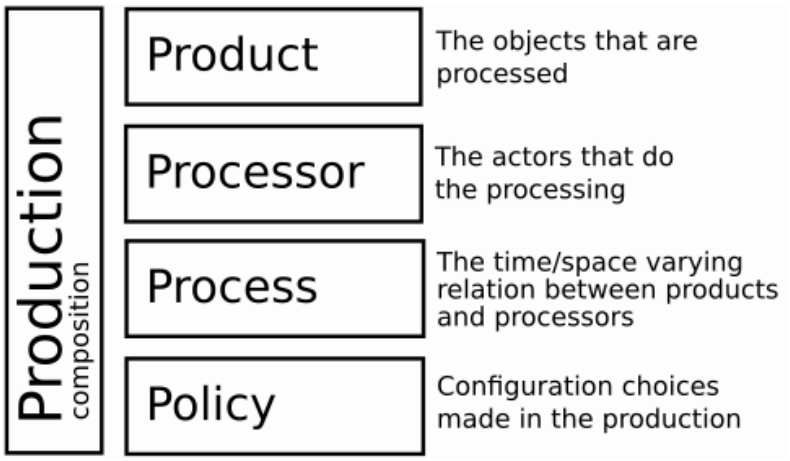


Figure 7: 5P model