# A Constraint Programming Approach to Optimize Network Calls by Minimizing Variance in Data Availability Times

Luis Neto[1], Henrique Lopes Cardoso[2], Carlos Soares[3], Gil Gonçalves[4]
{lcneto, hlc, csoares, gil}@fe.up.pt
Faculdade de Engenharia, Universidade do Porto, Porto, Portugal

[14]ISR-P, Instituto de Sistemas e Robótica - Porto, Portugal

[2]LIACC, Laboratório de Inteligência Artificial e Ciência de Computadores, Porto, Portugal

[3]INESC TEC, Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência, Porto, Portugal

***Abstract*—Smart Nodes are intelligent components of sensor networks that perform data acquisition and treatment, by performing virtualization of sensor instances. Smart Factories are an application domain in which dozens of these cyber-physical components are used, flooding the network with messages. In this work, we present a methodology to reduce the number of calls a Smart Node makes to the network. We propose grouping individual communications within a Smart Node to reduce the number of calls, which is important to improve the efficiency of the factory network. The paper exposes and explains the Smart Node internal structure, formally describing the problem of minimizing the number of calls Smart Nodes make to Cloud Services, by means of a combinatorial *Constraint Optimization Problem*. Using two *Constraint Satisfaction Solvers*, we have addressed the problem using distinct approaches. In this extended version of the work, an additional constraint is added to cut the search space, by eliminating infeasible solutions. Optimal and sub-optimal solutions for an actual problem instance have been found with both approaches. Furthermore, we present a comparison between both solvers in terms of computational efficiency, constraints created in the extended vs original version and show the solution is feasible to apply in a real case scenario.**

***Keywords–Sensor Simulation; Combinatorial Optimization; Time Synchronization; Smart Nodes; Industrial Wireless Sensor Networks.***

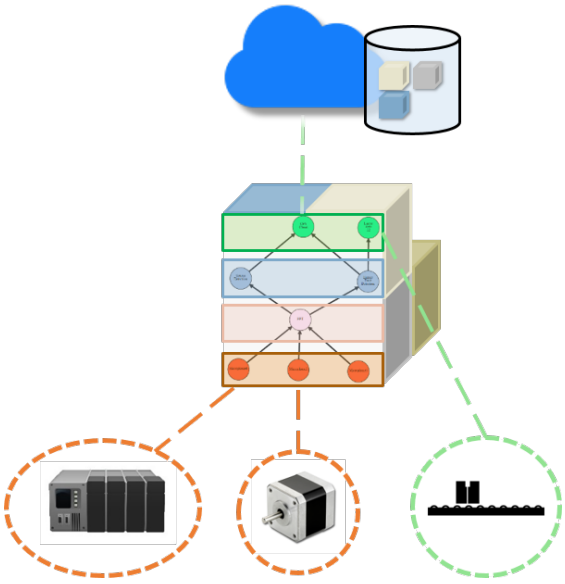

Figure 1. A Smart Node gateway.

## I. INTRODUCTION

Wireless Sensor Networks (WSN) consist of sensors sparsely distributed over a given area to sense physical properties, such as luminosity, temperature, current, etc. They are composed of sensor nodes, which pass data until a destination gateway is reached. Common applications are industrial and environment sensing, where they can be used to perceive the state of a machine and prevent natural disasters, respectively. Gateways in WSN play a preponderant role, since they acquire data from sensors, do pre-processing and are responsible to send sensors data to cloud systems for other forms advanced processing.

In this work we present an extension to a previous formulation [1] that solves the problem presented in the following sections. In this extended version, the problem was revised with the intent to increase the time efficiency of the previous proposed solution and also to explore in more detail the previous results. For this analysis, it was planned to run the same tests again, but for a longer period of time. After getting a second set of results, the problem was once again analysed. The analysis objective was try to find some constraint, formulation improvement or domain reduction that decreases the complexity for finding a solution.

The presented technology, a sensor gateway, inherits its main characteristics from the Smart Component philosophy. This philosophy is based in a consistent study of the Smart Manufacturing initiative and it is being systematically refined and matured by past and present European projects

(*XPress [2], IRamp3 [3], ReBORN [4] and SelSus [5]*). There are five essential characteristics to a Smart Component:

- **Reconfigurable and modular**: the solution must be capable to extend its capabilities by adding new software modules and it must be capable to reconfigure its internal operation in runtime.
- **Data processing capabilities**: system state assessment, event detection and fault alarm requires data processing capabilities.
- **Omnidirectional communication and interface capabilities**: omnidirectional means that the system must be capable to talk with devices at a lower level (sensors and machines), same level (other Smart Component's) and higher level (cloud servers, manufacturing systems).
- **Process events and take actions**: this capability provides the system with a certain degree of smartness and autonomy. In case any event of interest, the system must be capable of detecting it and take the proper actions.
- **Real-time acquisition, processing and delivering**: typically, field devices operate at variable real-time scales, performing multiple tasks in a coordinated way. Providing actions in real time is a vital factor for industrial scenarios.

What introduces the complexity that we are trying to address in the Smart Component is the fact that it was designed to be modular, according with Component-Based Software Engineering methodologies. It was developed as "a composite of sub-parts rather than a monolithic entity" [6]. The advantages of such tackle many objectives of the software industry, some of them are: reduction of production cost, code reuse, code portability, fast time to market, systematic approach to system construction and guided system design by formalization and use of domain specific modelling languages.

The component model is the foundation of a component based design. It defines, briefly, the composition standard, that is: how components are composed into larger pieces; how and if they can be composed at design and/or runtime phases of a component life-cycle; how they interact; how the component repository (if any) is managed and the runtime environment that contains the assembled application. Because all of this, component models are hard to build. Some known problems are: achieving deterministic and real-time characteristics; managing parallel flows of component and system development; maintaining components for reuse; different levels of granularity [7] and portability problems [8].

According to [6], components can be divided into 2 main classes: 1) objects, as in OO languages; 2) architectural units, that together compose a software architecture. According to the authors, there are no standard criteria for what constitutes a component model. Components syntax is the language used to component definition and which may be different from implementation language. Typically, the containers and runtime environments are designed and maintained in a server. In this case, we are dealing with an embedded system; being itself the runtime environment and container. The Smart Node uses architectural units as encapsulation for drivers that gather sensor and machine data, objects are used to implement algorithms for data treatment.

Figure 1 shows a Smart Node from an external operation of perspective. These components are nodes in Industrial Cyber Physical Systems that operate and control *Industrial Wireless Sensor Networks*. To introduce the problem this paper addresses, let us consider a scenario in which a reasonable number of these components operate simultaneously. In this operation the following conditions applies:

- Gateways are in constant synchronization with Intra/Inter Enterprise Cloud systems.
- Gateways perform collaborative tasks by talking over the network.
- Human Machine Interface devices proceed to on demand requests to the Smart Nodes.

A large quantity of messages is expected, generated by a large number of devices and services.

Gateways collect data from different sensor types (e.g., humidity, current, pressure). These cyber-physical components are coupled to industrial machines, along with several sensors, which collect data about the operation of machines; finally, the data collected is treated and synchronized with Cloud systems for multiple purposes. The majority of sensors coupled to industrial machines sample data at very different rates and synchronize the collected data with the Smart Node, in the respective sampling frequency. A Smart Node can embed a set of different data treatment modules. These modules can be instantiated to provide different ways of treating sensor data in a way that can be represented as a graph (Figure 2). A gateway internal logic arrangement is represented using a *directed acyclic graph (DAG)*. The graph structure in Figure 2 can be divided into three levels, each with a different label and colour assignment: the Sensor Level (bottom level), includes sensor instances providing data to the gateway; the Data Treatment Level (middle level), includes nodes representing instances of algorithms embedded at the gateway that can treat information in several ways (e.g., aggregate data using moving average, perform trend analysis or other functions); the Network Level (top level), includes nodes where the flow resulting from the lower level nodes can be redirected to subscribing hosts in the network. This internal structure can be dynamically rearranged: new sensors and data modules can be loaded into the Smart Node; the connections between nodes can be reformulated to synchronize and treat data in new ways.

A problem of efficiency emerges due to the different rates at which the data is gathered from all kinds of connected sensors. When data reaches the Network Level nodes, it is immediately sent to the subscriber, a node in the network, in this case, some cloud service. Slight time differences in the availability of data lead the different Network Level nodes to perform new and individual calls. If those time differences were eliminated, Network Level nodes would be synchronized and data from the different nodes could be packed together, reducing the total number of calls made and the network traffic heavily. To accomplish synchronization among Network Level nodes, data buffers for all the edges connecting nodes previous to a particular Network node must be resized to compensate: (1) different time to process data by Data Treatment level nodes, since each module takes different time to process data; (2) different sampling rates of sensors, a same number of samples is accumulated at different times.

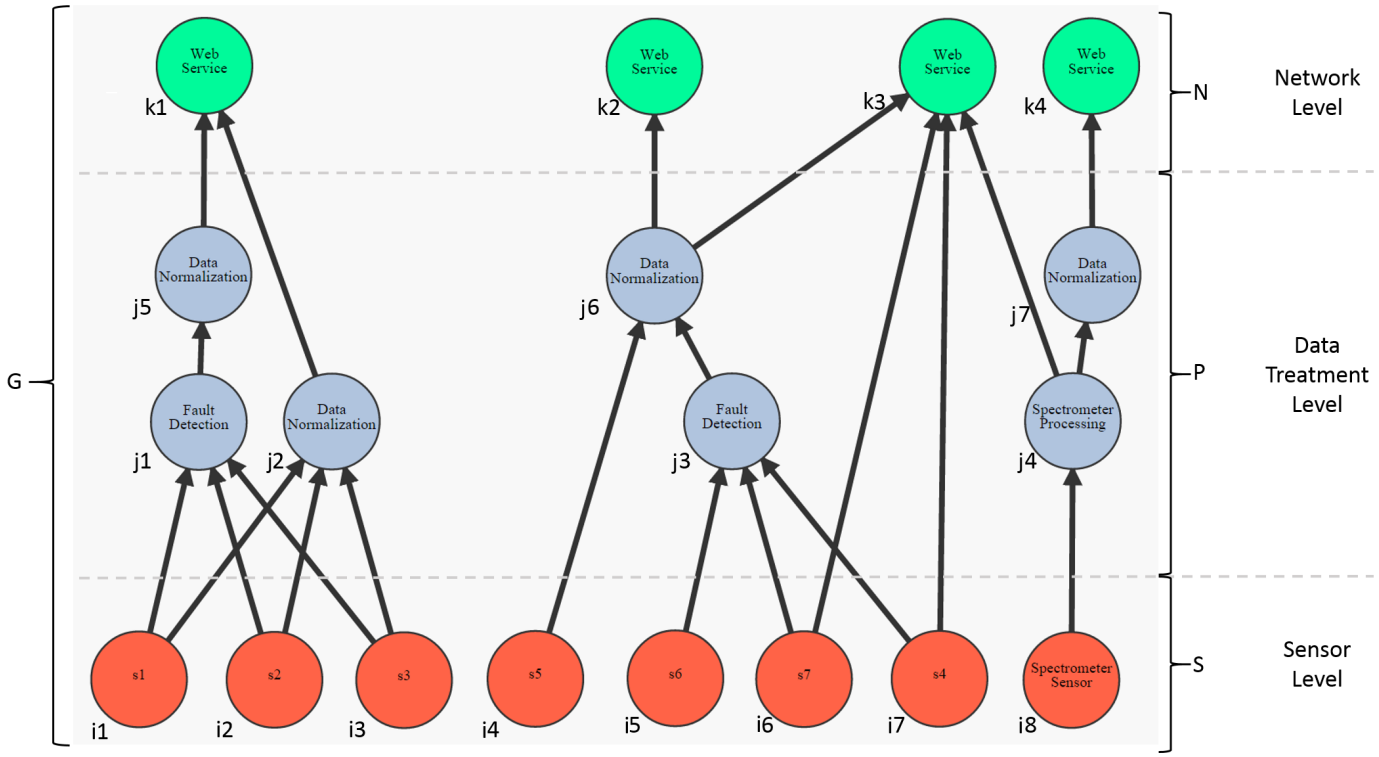


Figure 2. Internal Gateway configuration.

Taking advantage of the DAG representation of the gateway, we formulate and propose a solution to the problem as a combinatorial *Constraint Optimization Problem*.

The remaining chapters in this work describe the following: In Section II, a formal definition of the problem is presented. Section III shows literature review, the problem formulation basis. In Section IV, the solving process is detailed along with assumptions, constraints and technology that has been used. In Section V, the initial set of results is presented. Section VI explains the revision made to the initial problem solution, a new constraint is formally defined and the application of that constraint is reported in comparison with extended result sets. In section VII, conclusions and future work are described.

## II. Problem Definition

Each arc in the graph (see Figure 2) has an associated buffer $b_{n,m}$. Given the fact that sensors are sampling at different frequencies $freq$, these buffers are filled at different rates. We define $G$ as the set of nodes in a particular Gateway instance; three subsets of nodes are contained in $G$ : $N \subset G$ is the subset of Network Nodes (index $k$ nodes); $P \subset G$ is the subset of data Processing Nodes (index $j$ nodes); $S \subset G$ is the subset of Sensor Nodes (index $i$ nodes). The subsets obey to the following conditions:

$$G = N \cup P \cup S; N \cap P = \emptyset; P \cap S = \emptyset; N \cap S = \emptyset \quad (1)$$

Nodes in $N$ can be classified as consumers; nodes in $S$ are exclusively producers; nodes in $P$ are both producers and consumers. Edges between nodes can be defined as:

$$e_{n,m} = \begin{cases} 1 & \text{if } n \text{ is consumer of } m : n \neq m; \\ & m \in P \cup S \text{ and } n \in N \cup P \\ 0 & \text{otherwise} \end{cases} \quad (2)$$

As an example, we can observe in Figure 2 that node $j_6$ consumes from $i_4$ (Sensor Level) and $j_3$, which is in same level (Processing Level) and all the $k$ nodes (Network Level) only consume from inferior levels. To help in the definition of this problem, two additional subsets of nodes, containing the connections of a given node, are defined as follows:

$$W_n = \{j : j \in P \wedge e_{n,j} = 1\}, n \in N \cup P \quad (3)$$

Equation (3) defines a subset of nodes in $P$, which are producers for the given node $n \in N \cup P$. As an example (Figure 2), for $n = j_6 : W_{j_6} = \{j_3\}$; for $n = k_3 : W_{k_3} = \{j_6, j_4\}$; and for $n = j_3 : W_{j_3} = \emptyset$ since it does not consume from any Data Processing nodes.

$$X_n = \{i : i \in S \wedge x_{n,i} = 1\}, n \in N \cup P \quad (4)$$

Equation (4) defines a subset of nodes in $S$, which, are producers for the given node $n \in N \cup P$. In Figure 2, these are the nodes $i$ in the Sensor Level, from which, Processing Level nodes and Network Level nodes consume. As an example (Figure 2), for $n = k_3 : X_{k_3} = \{i_6, i_7\}$; for $n = k_1 : W_{k_1} = \emptyset$ since it does not consume from any Sensor Level node and for $n = j_6 : W_{j_6} = \{i_4\}$.

A processing node in $P$ applies an algorithm to transform the data coming from its associated producers. The data generated at the sensor level is delivered to the processing nodes as a batch, which contains the number of samples equal to the size of the buffer for the corresponding edge.

In order to the processing to be possible, the number of elements in each collection must be the same. This constraint must be applied to the subsets $W_n$ and $X_n$ of a given node $n$ in $N \cup P$, respectively; for that constraint to be respected, the size of every buffer associated to each element in $W_n \cup X_n$ must be the same. Formally this constraint can be represented as:

$$\forall n \in N \cup P, \forall m \in W_n \cup X_n : |b_{n,m}| = f(n) \quad (5)$$

Where $|b_{n,m}|$ represents the size of the given buffer for the given edge $e_{n,m}$ and $f(n)$ is the size of any buffer from which node $n$ consumes.

The size of a buffer is adjustable and can vary from 1 to 1000. The objective of this problem is to arrange a combination of values to parametrize the size of every buffer $|b|$, for every arc in the graph, which minimizes the differences between times at the Network Nodes in which data is available to be sent to the network. To calculate the time that takes data to be available at every node $k \in N$, the times for all its providers in the graph must be calculated. As data comes in collections (sets of single values), let us define *burst* as the exact time at which data is sent from one provider node to a consumer node and represent the *burst* of a node $n$ as $B_n$.

The *burst* of a Sensor Node $i$ is defined by the product of its sampling frequency and the size of the buffer associated to the edge $e_{n,i}$ we are assuming. That way, every time a sample from a sensor is collected, that sample is sent to all consumers of that sensor. A *burst* of a Sensor Node to an adjacent consumer node, $m$, occurs when the buffer for the edge $e_{i,m}$ is completely filled, and is formally represented by the expression:

$$B_{i,m} = freq(i) \times |b_{i,m}| \times e_{i,m} = 1; \forall i \in S \wedge m \in P \cup N \quad (6)$$

For a Data Processing Node, the burst time must contemplate all the burst times from its providers, the time that takes for the associated function $T(f(n))$ to treat one data sample and the size of the buffer associated to the edge $e_{n,m}$ we are assuming. The expression which determines the burst time for a Data Processing Node $j$ to a consumer node $m$ is defined as:

$$B_{j,m} = (\max_{i \in W_j \cup X_j}(B_{i,j}) + T(f_j) \times (|W_j| + |X_j|)) \times |b_{j,m}|; e_{j,m} = 1 \quad (7)$$

We assume that the growth in time complexity of the function $T(f_n) : n \in P$ is linear with the number of samples to process. Since the size of each producer buffer is equal, we multiply the total number of producers of $j$ by the cost of treating a single sample. To calculate the *burst* for $j_1$ (see Figure 2), we take the max *burst* of $X_{j_1}$ and sum the product of $T(f_{j_1})$ (time to process one sensor sample) with the number of elements in $X_{j_1}$ (which corresponds to the producers $i_1, i_2$ and $i_3$).

Finally, to calculate the *burst* of a Network Node $k \in N$:

$$B_k = \max_{i \in X_k \cup W_k}(B_{i,k}) \quad (8)$$

Using the expression to calculate the *burst* for each Network Node, the objective is to minimize the variance of *burst* for all the Network Nodes and also minimize the sum of all buffer sizes in the DAG. By varying the size of the buffers in the graph, the variance of all burst times for Network Nodes and the sum of all buffer sizes are minimized. With a variance of zero or closer, data from different Network Nodes can be packed in the same payload and sent to the subscribers in the network. Even if the quantity of data exceeds the maximum payload size for the protocol in use, or the physical link being used, the number of connections needed is far less than it is when using the original strategy of independent calls. The number of buffers $|P \cup N|$, times an upper bound buffer size of 1000 is multiplied by the variance. This way, the variance has more impact in the search of an optimal solution than the sum of all buffer sizes.

$$\hat{V}(B_k) \times \ 1000 \times |P \cup N| + \sum_{n \in |P \cup N|} \sum_{m \in W_n \cup X_n} |bn, m| \quad (9)$$

As follows from Equation (9), minimizing variance of burst times for network nodes is the major concern. To reflect this, the variance is multiplied by the maximum possible size for a buffer (1000, which is a reasonable number of samples for a sensor), times the number of Processing and Network nodes. This will drive the solver to focus on a solution with less variance, and break ties by considering the minimal buffer sizes (as these incur a cost). With a variance of 0 at the Network Level nodes, all data produced can be sent to the cloud using the same call. If variance is higher than 0, a threshold must be used to decide the maximum reasonable time to wait between bursts. In comparison with individual calls strategy – a call made every time a burst at the Network Level occurs – the number of calls to the cloud is minimized as a consequence. The theoretical search space of the problem is $E^n$, where $E$ represents the total number of edges in the graph and $n = 1000$ is the *Buffer Size* domain upper bound. The real search space, imposed by the constraint of Equation (5), can be determined by $F^n$, where $F = |P| + |N|$ is the total number of Processing and Network nodes in the graph.

## III. Related Work

To the best of our knowledge, there is no scientific literature or works that cover this exact problem. The problem presented in this work emerged due to the very specific nature of Smart Nodes applied to industrial monitoring situations. Since an exact formulation or solution to this problem could not be found, the related works presented are analogous in the sense that some knowledge could be used to refine the modelling and solutions presented.

The theoretical background behind this problem has a large spectrum of application. The problem of modelling buffer sizes is mostly applied to network routing, to which the works [9],[10] and [11] are examples. As we are not interested in dealing with networks intrinsic characteristics, those buffer

optimization problems can hardly be extrapolated to this work. The domain of Wireless Sensor Networks (WSN) is another scope of application of buffer modelling optimization, with relevant literature in this domain; the section of *Routing* problems in [12] covers a great number of important works regarding Flow Based Optimization Models, for data aggregation and routing problems. WSN optimization models care with constraints that this problem modulation does not cover, such as: residual energy of nodes, link properties, network lifetime, network organization and routing strategies.

A relevant work in WSN revealed to be of the major interest for this work. The authors presented and solved the problem of removing inconsistent time offsets, in time synchronization protocols for WSN [13]. The problem presented has a high degree of similarity with the case we are dealing. The problem is represented by a *Time Difference Graph (TDG)*, where each node is a sensor, every sensor has local time and every arc has an associated cost time given by a function. The solution to the problem is given by a Constraint Satisfaction Problem (CSP) approach. For every arc in the graph there exists an *adjustment variable* (analogous to the buffer size in this case), assignments are made to the variables to find the largest consistent sub-graph, i.e., a sub-graph in which inconsistent time offsets are eliminated.

Focusing the search in the literature domain of CSP problems, several works were revealed in the sub-domain of balancing, planning and scheduling activities that can be related to this application [14][15][16][17][18]. Namely, models of combinatorial optimization for minimizing the maximum/total lateness/tardiness of directed graphs of tasks with precedence and time constraints [14][18]. These problems are analogous to this work, and due to a simplified formulation with the same constraints (precedences and time between nodes), can be easily extrapolated to our case.

## IV. Implementing and Testing

### A. Problem Assumptions

The Smart Node application has several interfaces for real sensors, the physical connections range from radio frequency to cabled protocols. By testing this model with simulated scenarios, we assume no interference or noise of any type can cause disturbance in the sampling frequency. In a real case scenario, a sensor could enter in an idle state for a variety of reasons. In that case, data would not be transmitted at all, causing the transmission of data to the Cloud to be postponed for undefined time, waiting for the Network Level node burst depending on the idle sensor. For simplification, we assume a sensor never enters an idle state. Also, it is assumed that the time that takes to treat one sample of data will increase linearly for more than one sample, as mentioned for $T(f_j)$ when introducing Equation (7).

### B. Constraint Satisfaction Problem Solvers

For comparison of performance purposes we implemented the problem using both *OptaPlanner* and *SICStus Prolog*. As the Smart Node is implemented in Java we can take advantage of a direct integration with *OptaPlanner* in future. On the other hand, we expected that *SICStus Prolog* would produce the same results with better computation times because of the lightweight implementation and optimized constraint library. Using these premises and the results presented in the next section a grounded decision about what solver to use in future implementations of the Smart Node can be made.

### C. Tests

To validate the problem solutions, several DAG configurations were tested using the two implemented versions, based on *OptaPlanner* and *SICStus Prolog*, as described in Section IV. To test the implementations an algorithm to generate instances of the problem was built. The script generates instances of the Smart Node internal structure, DAG's, with a given number of Processing and Network nodes. Algorithm 1 briefly illustrates the approach:

**Data:** $G \leftarrow S \cup P \cup N$
**Result:** Smart Node internal configuration $G$
$notVisitedNodes \leftarrow G$;
$Pnodes \leftarrow randomInteger(\frac{|P \cup N|}{2}, |P \cup N| - 2)$;
$Nnodes \leftarrow nNodes - Pnodes$;
$Snodes \leftarrow$
$randomInteger(\frac{nNodes}{2}, nNodes + \frac{nNodes}{2})$;
$G \leftarrow S, P, N \leftarrow$
$generateNodes(Snodes, Pnodes, Nnodes)$;
$remainingEdges \leftarrow Pnodes \times 2 + Nnodes + Snodes$;
**while** $remainingEdges > 0$ **do**
  **if** $node \leftarrow notVisitedNodes.nextNode()$ **then**
    $notVisitedNodes.remove(node)$;
  **else**
    $node \leftarrow G.randomNode()$;
  **end**
  **if** *node is S* **then**
    connect to a random P or S node, disconnected nodes first;
    $remainingEdges - -$;
  **else if** *node is P* **then**
    get connection from a random P or S node, disconnected nodes first;
    connect to a random P or S node, disconnected nodes first;
    $remainingEdges - -$;
    $remainingEdges - -$;
  **else**
    get connection from random a P or S node, disconnected nodes first;
    $remainingEdges - -$;
  **end**
**end**

**Algorithm 1:** Smart Node instance generation.

Real scenarios generally have a higher number of Sensor Nodes, followed by a small number of Processing Nodes and an even smaller number of Network Nodes. Typically, the total number of nodes does not exceed 30 per operation. The instance generator picks aleatory numbers for the nodes bounded by a real case scenario application. Sampling frequencies for the sensors are assumed to vary from 400 to 2000 milliseconds. Functions to treat data in Processing Nodes are not typically complex. We measured the real case scenario functions to treat the minimum amount of data (1 sample) and we got values ranging from 0.19 to 0.38 milliseconds. To cover the buffer size domain, we need to take the worst case, 1000 samples. Given best and worst cases, the values attributed to cost of

Processing Nodes ( $T(f_j)$ in Equation (7)) are between 1 and 40 milliseconds.

*1) OptaPlanner:* This solver [19] is a pure *Java* constraint satisfaction *API* and solver that is maintained by the *RedHat* community. It can be embedded within the Smart Node application to execute and provide on-demand solutions to this optimization problem. Because of the reconfigurable property of the Smart Node internal structure, each time the structure is rearranged, the solution obtained to the problem instance prior to the reconfiguration becomes infeasible. The integration (see Figure 3) between the two technologies is accomplished by defining the problem in the OptaPlanner notation: (1) *Buffer Size* class corresponds to the *Planning Variable*, during the solving process it will be assigned by the different solver configurations; (2) *Edge* class is the *Planning Entity*, the object of the problem that holds the *Planning Variable*; (3) *SmartNodeGraph* class is the *Planning Solution*, the object that holds the problem instance along with a class that allows to calculate the score of a certain problem instance. The score is given by implementing Equation (9); the best hard score is 0, which corresponds to null variance between the Network Levels nodes. The soft score corresponds to the minimization of the sum of all buffer sizes and does not weight as much as a hard score in search phase.

Since the search space is exponential, heuristics can be implemented to help the *OptaPlanner* solver to determine the easiest buffers to change. The implemented heuristic sorts the buffers from the easiest to the hardest. The sorting values are given by the number of ancestors of a given edge, an edge with a greater number of ancestors is more difficult to plan. Also, if an edge leads to a Network Node, it is considered more difficult to plan. *OptaPlanner* offers a great variety of algorithms to avoid the huge search space of most *CSPs*. These algorithms can be consulted in the documentation [20] and configured to achieve best search performances. For a correct comparison we used the *Branch and Bound* algorithm, which is the same algorithm that *SICStus Prolog* uses by default, without heuristics.

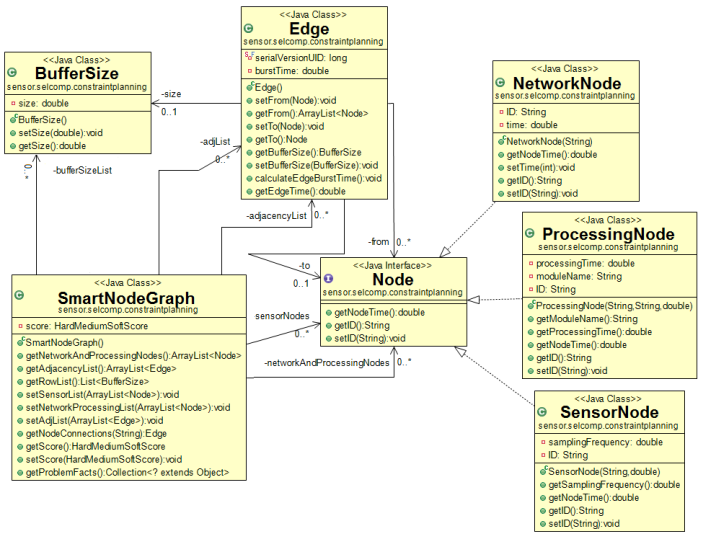


Figure 3. UML for Smart Node and OptaPlanner integration.

The UML diagram in Figure 3 shows the modelling of the problem using the *OptaPlanner* methodology.

*2) SICStus Prolog: SICStus Prolog* [21] provides several libraries of constraints that allow to model constraint satisfaction problems much more naturally than the *OptaPlanner* approach, which follows from the fact that modelling a problem in *SICStus Prolog* takes advantage of the declarative nature of logic programming. The problem modelling involved four types of facts (to represent $N$, $P$ and $S$ nodes, and to represent edges) and six predicates (to gather variables, express domain and constraints). The *clpfd* (Constraint Logic Programming over Finite Domains) [22] library was used to model and solve the problem. This library contains several options of modelling that can be used to optimize the labelling process. In our case, the labelling process takes as objective the minimization of the difference between the Network Node with the maximum burst time and the one with the lowest burst time (Equation (9)). The variables of the problem are given by a list of all the facts *edge(from,to,buffer_size).*, where *buffer_size* are the variables to solve in a finite domain from 1 to 1000. In future implementations of the problem, global constraints and labelling options must be analysed to ensure the modulation is the most optimized.

## V. RESULTS

For both implementations the first set of results is shown in Tables I and II. The results shown are an average of 5 different problem instances for each problem size, which is determined by $|P \cup N|$, see Section II. To gather results, the generator was used to generate 5 instances of the problem for each row. Then, both solvers were used in the same machine (Intel(R) Core(TM) i7-4710HQ CPU @ 2.50GHz (8 CPUs), 2.5GHz, 16384MB RAM), with the same conditions (Windows 10 Home 64-bit), to run the tests. We established a limit of 60s to run the tests, which was considered acceptable for the solvers to find a feasible solution in a real case. Another limit was the number of nodes used in the experiences. With a number of nodes in the order of 100, and a time window of 8 hours, both solvers were unable to give a response to most cases. Given the complexity stated, and the fact that in real cases the number of nodes normally does not exceed 30, 50 nodes was the limit used for the tests.

The quality of the solutions found is mostly given by the second column, which represents the constraint of minimizing the burst times at the Network Level. As we can be seen in Table II, the *SICStus Prolog* implementation shows the best results for the most relevant quality factor. In the third column, the sum of all the buffer sizes is lower in the *OptaPlanner* implementation (Table I). During the tests, it was observed in the logs that the *OptaPlanner* was much slower traversing the search tree. Regarding all the columns, a clear tendency to worst results is obvious along the table, but in the last line of both tables, a sudden improvement in the variance occurs. This behaviour enforces the NP-Completeness nature of this kind of problems. In every row of both tables, in which a *Solution time* of 60 seconds is found, that row matches a sub-optimal solution. Since both solvers were programmed to stop at 60 seconds, most solutions are not optimal. Sub-optimal solutions are feasible in a real case, even if the variance between call times is not zero, because the gap is heavily reduced. The

TABLE I. OptaPlanner results

| OptaPlanner | | | |
|---|---|---|---|
| $\|P \cup S\|$ | $\Delta V(B_k)$(ms) | $\sum \|b_{n,m}\|$ | *Solution time(s)* |
| 5 | 15.600 | 264.400 | 48.133 |
| 10 | 82.200 | 30.600 | 60.000 |
| 15 | 205.800 | 241.400 | 48.037 |
| 20 | 637.600 | 189.800 | 60.000 |
| 25 | 1494.000 | 56.600 | 60.000 |
| 30 | 1218.600 | 128.800 | 60.000 |
| 35 | 979.000 | 74.400 | 60.000 |
| 40 | 1434.400 | 74.600 | 60.000 |
| 45 | 1138.200 | 89.000 | 60.000 |
| 50 | 646.600 | 118.600 | 60.000 |

TABLE II. SICStus results

| SICStus | | | |
|---|---|---|---|
| $\|P \cup S\|$ | $\Delta V(B_k)$(ms) | $\sum \|b_{n,m}\|$ | *Solution time(s)* |
| 5 | 0.400 | 731.800 | 38.022 |
| 10 | 1.800 | 738.000 | 48.894 |
| 15 | 73.000 | 1738.600 | 60.000 |
| 20 | 102.400 | 4912.600 | 60.000 |
| 25 | 600.000 | 8210.800 | 60.000 |
| 30 | 457.800 | 6214.800 | 60.000 |
| 35 | 351.800 | 7986.200 | 60.000 |
| 40 | 564.000 | 5792.200 | 60.000 |
| 45 | 630.000 | 10457.000 | 60.000 |
| 50 | 321.200 | 14706.400 | 60.000 |

Smart Component can define a time window with the size of the variance, and this way, include all results in the same call.

## VI. Problem Revision

In this section, the second set of results is presented and discussed. A deeper problem analysis was conducted and the conclusions of this analysis are applied in a third set of tests.

Firstly, the time limit imposed to the search conducted in the previous results (Table I and Table II), was extended to 9 hours. To add more detail to the study, two new columns were added to the new results (Table III, Table IV and Table V). The initial variance $\Delta V_i(B_k)$, introduced in the first column, is the variance calculated with all buffer sizes set to the minimum size of 1. This column was introduced to give an idea of by how much is the time difference between the optimized version is produced and the first approach - set all buffers to same value, 1. The second column introduced was *"Total Search time (s)"*, it indicates the total amount of time that the search process took. For 32400 seconds (or 9 hours), it indicates a non optimal solution, because the search phase surpassed the time limit established. When the time indicated is the same as in *"Total Search time (s)"*, it means that an optimal solution was found timely. *"Total Search time (s)"* indicates the time that took to find the solution presented in the table, which is the last optimization found before the time limit was exceeded. With this separation it is possible to see that most of the results ($\Delta V_f(B_k)$) are found in a average time of 2.46 hours for the *SICStus* implementation. These results motivated a problem revision to decrease the search time and that is addressed in the following sections.

TABLE III. OptaPlanner Extended Results

| OptaPlanner | | | | | |
|---|---|---|---|---|---|
| $\|P \cup U\|$ | $\Delta V_i(B_k)$ (ms) | $\Delta V_f(B_k)$ (ms) | $\sum \|b_{n,m}\|$ | *Solution time (s)* | *Total Search time (s)* |
| 5 | 35 | 0 | 805 | 611.654 | 611.654 |
| | 1325 | 3 | 42 | 32400.000 | 32400.000 |
| 10 | 571 | 17 | 46 | 32400.000 | 32400.000 |
| | 1565 | 337 | 25 | 32400.000 | 32400.000 |
| 15 | 657 | 657 | 1122 | 32400.000 | 32400.000 |
| | 1313 | 520 | 29 | 32400.000 | 32400.000 |
| 20 | 320 | 51 | 1258 | 32400.000 | 32400.000 |
| | 1071 | 1071 | 43 | 32400.000 | 32400.000 |
| 25 | 1833 | 1833 | 44 | 32400.000 | 32400.000 |
| | 332 | 332 | 53 | 32400.000 | 32400.000 |
| 30 | 1553 | 1553 | 55 | 32400.000 | 32400.000 |
| | 1192 | 1192 | 52 | 32400.000 | 32400.000 |
| 35 | 2096 | 2096 | 56 | 32400.000 | 32400.000 |
| | 464 | 464 | 89 | 32400.000 | 32400.000 |
| 40 | 251 | 251 | 106 | 32400.000 | 32400.000 |
| | 464 | 464 | 89 | 32400.000 | 32400.000 |
| 45 | 6 | 0 | 98 | 12896.000 | 32400.000 |
| | 365 | 365 | 64 | 32400.000 | 32400.000 |
| 50 | 1364 | 1364 | 112 | 32400.000 | 32400.000 |
| | 496 | 496 | 101 | 32400.000 | 32400.000 |

TABLE IV. SICStus Extended Results

| SICStus | | | | | |
|---|---|---|---|---|---|
| $\|P \cup U\|$ | $\Delta V_i(B_k)$ (ms) | $\Delta V_f(B_k)$ (ms) | $\sum \|b_{n,m}\|$ | *Solution time (s)* | *Total Search time (s)* |
| 5 | 35 | 0 | 805 | 40.594 | 40.594 |
| | 1325 | 0 | 83 | 192.076 | 192.076 |
| 10 | 571 | 24 | 4059 | 6609.570 | 6609.570 |
| | 1565 | 42 | 1420 | 32400.000 | 32400.000 |
| 15 | 657 | 87 | 1122 | 19933.302 | 32400.000 |
| | 1303 | 102 | 3834 | 1867.245 | 32400.000 |
| 20 | 320 | 0 | 1258 | 1183981.000 | 11839.810 |
| | 1071 | 2 | 3666 | 32400.000 | 32400.000 |
| 25 | 1833 | 0 | 1258 | 32400.000 | 32400.000 |
| | 332 | 2 | 3666 | 655.428 | 32400.000 |
| 30 | 1553 | 44 | 49 | 32400.000 | 32400.000 |
| | 1192 | 397 | 741 | 3564.869 | 32400.000 |
| 35 | 2096 | 661 | 380 | 2.409 | 32400.000 |
| | 464 | 464 | 89 | 631.403 | 32400.000 |
| 40 | 251 | 44 | 7398 | 2341415.000 | 32400.000 |
| | 464 | 191 | 15790 | 29237.796 | 32400.000 |
| 45 | 6 | 0 | 12387 | 9862.394 | 32400.000 |
| | 365 | 102 | 1459 | 11052.058 | 32400.000 |
| 50 | 1364 | 922 | 8630 | 7666.592 | 32400.000 |
| | 496 | 15 | 19318 | 150.605 | 32400.000 |

### A. Proposed Enhancement

To analyse the problem more deeply, let us consider the buffers in the graph were adjusting its size is really critical. The buffers in inferior levels, all that connect $P$ and $S$ nodes,

although their size may impact the final solution, are less critical. The unique constraint that must hold on these is the one of Equation (5). This constraint implies that buffers connecting $P$ nodes must have the same size. In these cases, the impact on the superior level is dictated by the burst time of the latest provider to send data. There is nothing that can be done beyond this constraint. On the other hand, let us consider buffers that connect directly to network nodes $N$. These buffers dictate the optimization function defined in Equation (9). By increasing the buffer size, we are multiplying it by the burst time of the latest consumer to provide data. That is to say, if the values of these buffers are different, we cannot multiply the same value on two or more different buffers. If the same value is multiplied, we are maintaining the difference between them. Taking this in consideration, there is an obvious constraint to apply in this case, constrain buffers with different burst times of having the same size. Although this seems a little improvement, the impact of this constraint grows in function of the number of different buffers being considered.

To formalize this constraint, let us first define a set that contains the buffers of all edges that connect directly Network Nodes. Let us denote this set by $NBs$, which stands for "Network Buffer's". We can define this set recurring to the previous set definitions in Equation (3) and Equation (4), as follows:

$$NBs = \{b_{k,m} : \forall k \in N, m \in X_k \cup W_k\} \tag{10}$$

Relying on the DAG of Figure 2 to give a clearer example, this set would contain the following buffers $NBs = \{b_{k_1,j_5}, b_{k_1,j_2}, b_{k_2,j_6}, b_{k_3,j_6}, b_{k_3,i_6}, b_{k_3,i_7}, b_{k_3,j_4}, b_{k_4,j_7}\}$.

Having at this point a clear view of types of buffers that will be the target of this optimization, we can introduce the constraint that will be only applied when the buffers belong to edges were the following conditions apply. The first condition for applicability of this constraint is that it can only be applied to buffers with different burst times ($B_{i,n_1} \neq B_{j,n_2}$). The second condition is that the source of data cannot be the same, because implicitly the burst time will be the same. Considering Figure 2, the buffers $b_{k_2,j_6}$ and $b_{k_3,j_6}$ fall in these two conditions. They have the same source, implicitly they have the same burst time. The logic of this constraint is: if we multiply the same factor (buffer size) by the same value (burst time), we are maintaining the variance between the two buffers being considered.

$$\forall i \in BNs, \forall j \in BNs, i \neq j, B_{i,n_1} \neq B_{j,n_2}, n_1 \neq n_2 : |b_{i,n_1}| \neq |b_{i,n_2}| \tag{11}$$

The impact of this constraint can be theoretically calculated for the worst search case, i.e., explore all the possible combinations of buffers sizes in $NBs$. Let us define the number of possible combinations for buffer sizes as $1000^B$. In which 1000 is the domain size for a buffer and $B$ is the number of buffers in $NBs$. Now we need to obtain the number of buffers that correspond to edges with different burst times. If we apply the conditions of Equation (11) to $NBs$, specifically the part that guarantees different burst times ($B_{i,n_1} \neq B_{j,n_2}$), we obtain the number of buffers $D$ to which this constraint can be applied. Considering the previous definitions for $B$ and $D$, by relying on combinatorics, we can apply simple arrangements to calculate the number of times that two ore more buffers in a search assignment will be equal. By subtracting the number of combinations cut from the search space (due to the application of Equation (11)) to the total number of assignments, we get the optimized number of possible combinations.

$$1000^B - \frac{1000!}{(1000 - D)!} \tag{12}$$

Considering the most basic case, five buffers ($B = 5$), and only two different among them ($D = 2$), this would give us a reduction in the search space of 999000 possibilities. In the most optimistic case, in which all buffers are different ($B = 5$ and $D = 5$), the reduction in search space is exponentially best, resulting in a cut of $9.90034950024 \times 10^{14}$ possibilities.

### B. Enhancement Tests

This subsection reports the results of implementing the optimization constraint developed in the previous section. The same conditions (hardware and time limit) and problem instances (same graphs) as in the previous tests were used. Because of the results obtained by the *OptaPlanner* implementation (Table III), the optimization constraint was only implemented in the *SICStus* solution. Table V shows the results for optimized version of the problem.

TABLE V. SICStus Enhancement Results

| SICStus Optimized | | | | | |
|---|---|---|---|---|---|
| $\|P \cup U\|$ | $\Delta V_i(B_k)ms$ | $\Delta V_f(B_k)ms$ | $\sum \|b_{n,m}\|$ | *Solution time (s)* | *Total Search time (s)* |
| 5 | 35 | 0 | 805 | 1.284 | 1.284 |
| | 1325 | 0 | 83 | 57.004 | 57.004 |
| 10 | 571 | 2 | 424 | 3526.065 | 32400.000 |
| | 1565 | 42 | 1420 | 2.817 | 32400.000 |
| 15 | 657 | 182 | 75 | 491.078 | 32400.000 |
| | 1313 | 163 | 490 | 1438.775 | 32400.000 |
| 20 | 320 | 1 | 261 | 2282.241 | 32400.000 |
| | 1071 | 278 | 3492 | 71.76 | 32400.000 |
| 25 | 1833 | 797 | 3677 | 7358.274 | 32400.000 |
| | 332 | 8 | 204 | 322.932 | 32400.000 |
| 30 | 1553 | 388 | 285 | 1653.479 | 32400.000 |
| | 1192 | 351 | 7002 | 1396.515 | 32400.000 |
| 35 | 2096 | 5129 | 301 | 213.453 | 32400.000 |
| | 464 | 330 | 9827 | 382.596 | 32400.000 |
| 40 | 251 | 216 | 948 | 2379.582 | 32400.000 |
| | 464 | 944 | 2001 | 12295.929 | 32400.000 |
| 45 | 6 | 0 | 121 | 556.953 | 556.953 |
| | 365 | 508 | 827 | 7684.82 | 32400.000 |
| 50 | 1364 | 20771 | 495 | 79.019 | 32400.000 |
| | 496 | 78 | 1365 | 110.605 | 32400.000 |

In Table VI, a comparison between the number of prunings for the same iterations and versions (in each table, for the same number of nodes $|P \cup U|$, there are two rows for the two different problem graphs tested) of the test is presented. The number of prunings was obtained by SICStus, using the *fdstats* predicate. As can be verified, the number of prunings was increased, which means that the improvement introduced is reducing the search space by cutting the search tree more times in the optimized version of the implementation.

TABLE VI. SICStus Enhancement Statistics

| | $\|P \cup U\|$ | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Prunings | 1st. Optimized | 43192808 | 4.122E+10 | 1.23E+10 | 6.24E+10 | 6.76E+10 | 4.78E+10 | 1.69E+11 | 4.14E+10 | 3.8E+10 | 2.78E+10 |
| | 1st. Non Opt. | 74079794 | 151387289 | 5.67E+08 | 3.04E+10 | 2.03E+10 | 1.14E+10 | 3.97E+10 | 1.67E+10 | 6.64E+10 | 2.78E+10 |
| | 2nd. Optimized | 51587328 | 4.423E+10 | 2.58E+10 | 2.13E+10 | 2.78E+10 | 2.84E+10 | 6.3E+10 | 6.58E+10 | 7.11E+10 | 3.19E+10 |
| | 2nd. Non Opt | 45262552 | 539864744 | 2.29E+10 | 2.99E+10 | 3.01E+10 | 2.05E+10 | 2.39E+10 | 4.44E+10 | 3.06E+10 | 4.21E+10 |

Despite the optimized version having reached faster solutions in practically all cases, as shown in the graph of Figure 4, the solution quality was affected negatively as can be seen in the graph of Figure 5. Although this might seem a worse strategy at first sight, it will always guarantee that the ideal solution of $\Delta V_f(B_k) = 0$ is found faster than in the previous implementation.

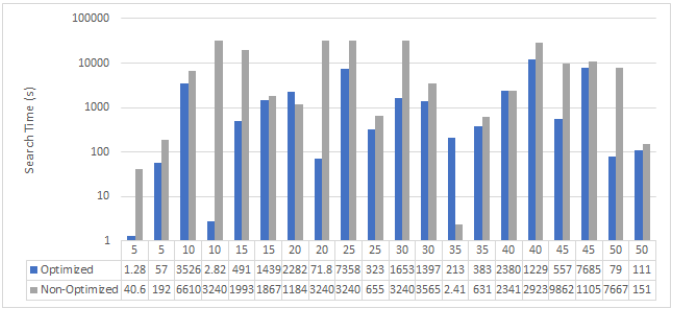


Figure 4. Search Time Comparison, Optimized vs Non-Optimized

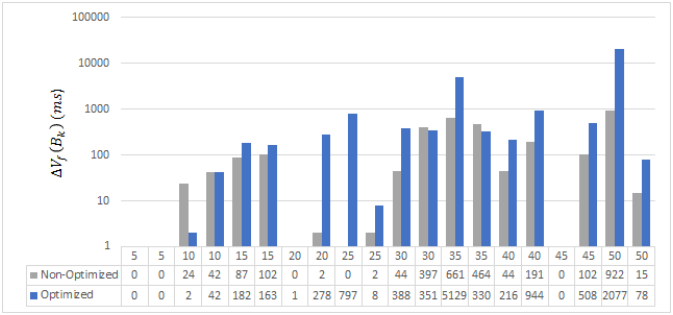


Figure 5. Solution Quality Comparison, Optimized vs Non-Optimized

## VII. Conclusion and Future Work

Despite the search space of the problem, both solvers reached optimal solutions in cases that are feasible to real application. In the future tuning options of the solvers must be explored. Another additional constraint to this problem could be the introduction of a case in which a single or several sensors are producing data with a higher priority. The problem can be easily reformulated to embrace that kind of situation by modifying the objective function Equation (9). *SICStus Prolog* shows a clear advantage in computation time. That difference can be the reflex of the number of code lines needed to model the problem. *SICStus Prolog* required eight procedures (predicates), against 10 classes and 1 *XML* configuration file for the *OptaPlanner* implementation. The difference in modelling complexity possibly causes an additional overhead. Another important remark is that, given the experience of implementing the problem and playing with the solvers options, two contrasts can be highlighted: (1) *SICStus Prolog* is very intuitive at the problem modelling phase, on the other hand, *OptaPlanner* required more effort, both in implementing an perceiving the methodology; (2) tuning the solvers, for example the time out feature that allows to stop the solver in the desired time, it is more intuitive in the *OptaPlanner* approach.

Regarding the optimization presented, the solution quality suffers with the constraint proposed in Equation (11). This decrease in quality of the solution is due to the fact that buffers involved cannot be equal. Despite achieving a worse variance, in cases were it is possible to achieve the ideal solution of zero variance, this implementation will find it faster, as shown in Figure 5. In this case, there exists a trade-off between better sub-optimal solutions (the non optimized version) and better chance to find optimal solutions (optimized version).

Considering all pros and cons, *SICStus Prolog* most probably will be chosen to integrate the Smart Node in future work. These experiments were made off-line, as future work, the Smart Component can embed the optimization code and adopt a strategy to optimize the variance in idle CPU time until an optimal solution is found on-line. In this extended version can be verified that, when the problem is too big, the complexity outperforms a reasonable time for a solution. As future work, an idea to split the DAG in sub-graphs, arrange individual solutions, and later join them using intermediary buffers.

## Acknowledgment

SelSus EU Project (FoF.NMP.2013-8) Health Monitoring and Life-Long Capability Management for SELf-SUStaining Manufacturing Systems funded by the European Commission under the Seventh Framework Programme for Research and Technological Development.

## References

[1] L. Neto, H. L. Cardoso, C. Soares, and G. Gonçalves, "Optimizing network calls by minimizing variance in data availability times," in Proceedings of INTELLI 2016 : The Fifth International Conference on Intelligent Systems and Applications (includes InManEnt 2016). IARIA, 2016, pp. 142–147.

[2] M. Peschl, N. Link, M. Hoffmeister, G. Gonçalves, and F. L. Almeida, "Designing and implementation of an intelligent manufacturing system," Journal of Industrial Engineering and Management, vol. 4, no. 4, 2011, pp. 718–745.

[3] G. Gonçalves, J. Reis, R. Pinto, M. Alves, and J. Correia, "A step forward on intelligent factories: A smart sensor-oriented approach," in Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA). IEEE, 2014, pp. 1–8.

[4] R. Fomseca, S. Aguiar, M. Peschl, and G. Gonçalves, "The reborn marketplace: an application store for industrial smart components," in INTELLI 2016 : The Fifth International Conference on Intelligent Systems and Applications (includes InManEnt 2016). IARIA, Nov. 2016, pp. 136–141.

[5] L. Neto, J. Reis, D. Guimaraes, and G. Goncalves, "Sensor cloud: Smartcomponent framework for reconfigurable diagnostics in intelligent manufacturing environments," in Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on. IEEE, 2015, pp. 1706–1711.

[6] K.-K. Lau and Z. Wang, "Software component models," IEEE Transactions on software engineering, vol. 33, no. 10, 2007, pp. 709–724.

[7] C. Maga, N. Jazdi, and P. Göhner, "Reusable models in industrial automation: experiences in defining appropriate levels of granularity," IFAC Proceedings Volumes, vol. 44, no. 1, 2011, pp. 9145–9150.

[8] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel, "A dynamic component model for cyber physical systems," in Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering. ACM, 2012, pp. 135–144.

[9] I. Ioachim, J. Desrosiers, F. Soumis, and N. Bélanger, "Fleet assignment and routing with schedule synchronization constraints," European Journal of Operational Research, vol. 119, no. 1, 1999, pp. 75–90.

[10] K. Avrachenkov, U. Ayesta, E. Altman, P. Nain, and C. Barakat, "The effect of router buffer size on the tcp performance," in In Proceedings of the LONIIS Workshop on Telecommunication Networks and Teletraffic Theory. Citeseer, 2001.

[11] K. Avrachenkov, U. Ayesta, and A. Piunovskiy, "Optimal choice of the buffer size in the internet routers," in Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on. IEEE, 2005, pp. 1143–1148.

[12] A. Gogu, D. Nace, A. Dilo, and N. Meratnia, Review of optimization problems in wireless sensor networks. InTech, 2012.

[13] M. Jadliwala, Q. Duan, S. Upadhyaya, and J. Xu, "On the hardness of eliminating cheating behavior in time synchronization protocols for sensor networks," Technical Report 2008-08, State University of New York at Buffalo, Tech. Rep., 2008.

[14] J. Błazewicz, W. Kubiak, and S. Martello, "Algorithms for minimizing maximum lateness with unit length tasks and resource constraints," Discrete applied mathematics, vol. 42, no. 2, 1993, pp. 123–138.

[15] B. Gacias, C. Artigues, and P. Lopez, "Parallel machine scheduling with precedence constraints and setup times," Computers & Operations Research, vol. 37, no. 12, 2010, pp. 2141–2151.

[16] K. Rustogi et al., "Machine scheduling with changing processing times and rate-modifying activities," Ph.D. dissertation, University of Greenwich, 2013.

[17] A. Malapert, C. Guéret, and L.-M. Rousseau, "A constraint programming approach for a batch processing problem with non-identical job sizes," European Journal of Operational Research, vol. 221, no. 3, 2012, pp. 533–545.

[18] J. H. Patterson and J. J. Albracht, "Technical noteassembly-line balancing: Zero-one programming with fibonacci search," Operations Research, vol. 23, no. 1, 1975, pp. 166–172.

[19] O. Team, OptaPlanne - Constraint Satisfaction Solver, Red Hat, (Last accessed 09-May-2017). [Online]. Available: http://www.optaplanner.org/

[20] ——, OptaPlanner User Guide, Red Hat, (Last accessed 09-May-2017). [Online]. Available: http://docs.jboss.org/optaplanner/release/6.3.0.Final/optaplanner-docs/pdf/optaplanner-docs.pdf

[21] M. Carlsson, J. Widen, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland, SICStus Prolog user's manual. Swedish Institute of Computer Science Kista, Sweden, 1988, vol. 3, no. 1.

[22] M. Carlsson, G. Ottosson, and B. Carlson, "An open-ended finite domain constraint solver," in Programming Languages: Implementations, Logics, and Programs. Springer, 1997, pp. 191–206.

[23] I. P. Gent, K. E. Petrie, and J.-F. Puget, "Symmetry in constraint programming," Foundations of Artificial Intelligence, vol. 2, 2006, pp. 329–376.