A Constraint Programming Approach for Managing End-to-end Requirements in Sensor Network Macroprogramming

Farshid Hassani Bijarbooneh[1], Animesh Pathak[2], Justin Pearson[1],
Valerie Issarny[2], and Bengt Jonsson[1]
[1] *Uppsala University, Department of Information Technology, Box 337, SE-75105, Uppsala, Sweden*
[2] *INRIA Paris Rocquencourt, France*
{*Farshid.Hassani, Justin.Pearson, Bengt.Jonsson*}*@it.uu.se,* {*Animesh.Pathak, Valerie.Issarny*}*@inria.fr*

Keywords: sensor networks, macroprogramming, constraint programming, non-functional requirements, task mapping

Abstract: Though several high-level application development (macroprogramming) approaches have been proposed in literature for wireless sensor networks (WSN), there is a need to enable support for expressing and supporting end-to-end non-functional constraints such as latency in WSN macroprograms. We augment an existing macroprogramming language and its compilation process to enable the specification of end-to-end requirements, and propose task mapping algorithms to satisfy those requirements through a constraint programming approach. Through evaluations on realistic application task graphs, we show that our constraint programming model can effectively capture the end-to-end requirements and efficiently solves the combinatorial problem introduced.

1 INTRODUCTION

Sensor network macroprogramming refers to the development of Wireless Sensor Network (WSN) applications at the system level (as opposed to at the node level), and holds the promise of enabling not just systems programmers (as is the case today) but also *domain experts* (e.g., architects, biologists, city planners etc.) to develop WSN applications. A sizeable body of work exists in this area [Mottola and Picco, 2011], with programming styles ranging from Haskell-like functional languages [Newton et al., 2007] to Python-like imperative ones [Gummadi et al., 2005].

To complement the above, algorithms have been developed to convert the high level descriptions to node-level code by solving problems such as optimal task-mapping in order to satisfy non-functional properties. While existing approaches focus on *system-wide* or *node-level* goals, such as minimising total energy consumption or minimising the maximum energy consumed by any node, work on managing *end-to-end* requirements (e.g., minimising the latency between the sensing of a phenomenon and the resulting action) in WSN macroprogramming is largely missing.

The focus of this paper is precisely the above kind of scenarios, where the developer is interested in minimising an end-to-end property while mapping tasks on the nodes in a particular sensor network deployment. A classic case would be a highway traffic management application (e.g., the one used in [Mottola et al., 2007]), where local regulations might need a particular deployment to provide guarantees between, say, the sensing of a traffic jam (speed = 0) and the actuation of the ramp signals to red in order to stop more cars from coming in. Similar latency restrictions apply in the case of, for example, traffic lights controlling access to a railroad crossing [US Department of Transportation, Federal Highway Administration, 2012].

We base our work on the Abstract Task Graph (ATaG) macroprogramming language [Bakshi et al., 2005b], based on the *data-driven macroprogramming* paradigm, where the developer breaks up the functionality of their application into *tasks* that interact with each other only using the *data items* that they produce and consume, and do not share any state otherwise. This technique is shown to be especially useful in specifying a wide range of sense-and-respond applications [Pathak et al., 2007]. ATaG includes an extensible, high-level programming model to specify the application behaviour, and a corresponding node-level run-time support, the data-driven ATaG runtime (DART) [Bakshi et al., 2005a]. The compilation of ATaG programs consists of mapping the high-level

ATaG abstractions to the functionality provided by DART, thus involving a task-mapping process. While the ATaG compiler currently supports task-mapping while optimising node-level energy properties, it does not have support for end-to-end properties such as latency.

The main related work to ours is [Pathak and Prasanna, 2010], where the authors optimised task-mapping for energy-related metrics using an MIP, and [Hassani Bijarbooneh et al., 2011], which used constraint programming (CP) [Rossi et al., 2006] to solve this. In [Tian et al., 2005], the authors only investigate *single*-hop *homogeneous* WSNs. We consider *multi*-hop *heterogeneous* networks including routing costs. To the best of our knowledge, there has been no work addressing a CP approach for a general case of task mapping in a multi-hop WSN achieving end-to-end latency optimisation under routing costs. Works such as the one by Wu et. al., in [Wu et al., 2010] are complementary to this, since they perform *runtime* evaluation of the WSN, while we focus on providing feedback to the developer before deployment. Note that the task mapping problem with end-to-end requirements is different from those encountered in traditional distributed systems such as clusters, since in our work, there is a strong relationship between the physical regions and the sensor nodes, as well as the constraints enforced by the end-to-end requirements between the tasks and the sensor nodes.

The main contributions of this paper are *i)* The formulation of the task-mapping problem arising out of the need to satisfy end-to-end constraints in WSN macroprogramming; *ii)* Constraint programming techniques to solve the latency problem; and *iii)* Extension of the approach to address situations where replication of tasks is allowed to improve latency performance.

We provide background on the macroprogramming language used in Section 2, followed by the problem definition in Section 3 and the mathematical formulation of the optimisation goal in Section 4. Section 5 presents our CP-based approach, and Section 6 discusses the results of our evaluation on a realistic traffic-management application. Section 7 concludes.

2 BACKGROUND

For completeness, in this section we provide some background on the programming model and compilation process of ATaG programs, as well as the modifications we have made to the language. For more complete details of ATaG, we refer the readers to the work in [Pathak and Prasanna, 2011].

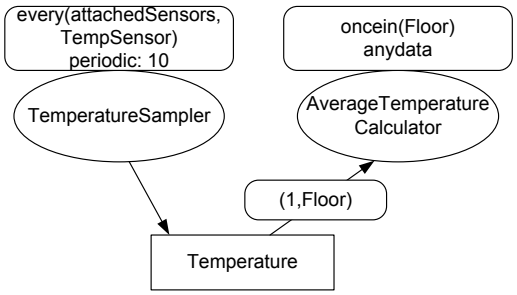

Figure 1: ATaG program for data-gathering

2.1 Programming Model

ATaG provides a data driven programming model and a mixed *imperative-declarative* program specification. Developers use a *data driven* model for specifying reactive behaviours, and *declarative specifications* to express the placement of processing locations and the patterns of interactions.

The overall structure of computation in an ATaG program is expressed as a task graph (see Figure 1 for an example), which consists of the following:

- **Abstract Data Items:** These represent the information in the various stages of its processing through the deployed application.
- **Abstract Tasks:** These are software entities that consume and produce instances of abstract data items. To specify their location and invocation patterns, they are annotated with *instantiation rules* and *firing rules* respectively.
- **Abstract Channels:** Abstract tasks and data items are connected through abstract channels, which are further annotated with logical scopes [Mottola et al., 2007], expressing the interest of tasks in only certain instances of data items.

In addition to the above, the developer specifies the internal functionality of each abstract task in an ATaG program using an imperative language such as Java. To interact with the underlying runtime system, each task must implement a **`handleDataItemReceived()`** method for each type of data item that it is supposed to process. The task can output its data by calling the **`putData()`** method implemented by the underlying runtime system.

Figure 1 illustrates an example ATaG program specifying a data gathering application [Choi et al., 2004] for building environment monitoring. Sensors within a cluster take periodic temperature readings, which are then collected by the correspond-

ing cluster-head. The *TemperatureSampler* task represents the sensing in this application, while the *AverageTemperatureCalculator* task takes care of the collection and computing the average. The tasks communicate through the *Temperature* data item. The *TemperatureSampler* is triggered every 10 seconds according to the **`periodic`** firing rule. The **`any-data`** rule requires *AverageTemperatureCalculator* to run when a data item is ready to be consumed on *any* of its incoming channels. The **`every(<property>,<value>)`** instantiation rule requires the task to be instantiated on each node where the particular property has that value. E.g., the *TemperatureSampler* task in the example will be instantiated on every node equipped with a temperature device. Since the programmer requires a single *AverageTemperatureCalculator* to be instantiated on every floor in the building, the **`oncein(Floor)`** instantiation rule is used for this task. Its semantics is to derive a system partitioning based on the values of the node attribute provided (**`Floor`**).

The **`(1,Floor)`** annotation on the channel specifies a number of hops counted in terms of how many system partitions can be crossed, independent of the physical connectivity. Since *Temperature* data items are to be used within *one* partition (floor) from where they generated, they will be delivered to *AverageTemperatureCalculator* instances running on the same floor as the task that produced them, as well as adjacent floors.

2.2 Compilation Process

In the previous section, we described the ATaG data-driven macroprogramming paradigm. In this section, we provide a formal definition of the process of compiling data-driven macroprograms to node-level code using the application given in Figure 1 as an example.

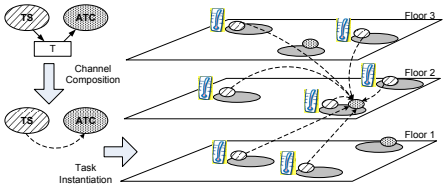

Figure 2: An example illustrating the compilation process of our sample program. Task names have been abbreviated.

2.2.1 Input

The input to the compilation process consists of the following three components.

Abstract Task Graph (Declarative Part): Formally, an abstract task graph $A(AT, AD, AC)$ consists of a set AT of abstract tasks and a set AD of abstract data items. The set of abstract channels AC can be divided into two subsets – the set of *output channels* $AOC \subseteq AT \times AD$ and a set of *input channels* $AIC \subseteq AD \times AT$.

Imperative Code for Each Task: As mentioned earlier, each abstract task is accompanied by code that details the actions taken when data is received by instances of that task.

Network Description: For every node n in the target network N, the compiler is also given the list of sensors and actuators attached to the sensor node n, as well as, a set of $(RegionLabel, RegionID)$ attribute-value pairs to denote the membership of the sensor node n in the regions of the network (e.g., $\{(\text{Floor}, 5), (\text{Room}, 2)\}$).

Runtime Library Files: These files contain the code for the basic modules of the runtime system that are not changed during compilation, including routing protocols etc.

2.2.2 Output

The compilation process generates a distributed application for the target network description commiserate with what the developer specified in the ATaG program. The output consists of the following parts:

Task Assignments: This is a mapping between the instances of the members of AT onto the nodes in N, in accordance with their instantiation rules.

The compiler must decide on the mapping to allocate the instantiated copies of the abstract tasks in AT to the nodes in N so as to satisfy all placement constraints specified by the developer.

Customised Runtime Modules: The compiler must customise the *DataPool* of each node to contain a list of the data items produced or consumed by the tasks hosted by it. It also needs to configure the *ATaGManager* module with a list of composed channel annotations, so when a data item is produced, the runtime can compute the constraints imposed on the nodes which are hosting the recipient tasks for it.

Cost Estimates: The compiler also reports on the non-functional aspects of the final deployed application, informing the developer of the *cost* of executing application A on a network N. Note that the actual nature of the cost estimates returned can vary depending on the developer's needs. This is the place where this paper builds upon the existing work, by bringing end-to-end cost estimates also in the compiler's purview.

2.2.3 Process Overview

The overall process takes place in the following steps:

Composition of Abstract Channels: In this step, each *path* $AT_i \rightarrow AD_k \rightarrow AT_j$ in the abstract task graph is converted to an *edge* $AT_i \rightarrow AT_j$, resulting in a graph with only task-task edges, keeping the annotation of the channel $AD_k \rightarrow AT_j$.

Instantiating Abstract Tasks: The *instantiated task graph* (ITaG) consists of instances of the tasks defined in AT, connected to each other according to the channel annotations in A. e.g., due to the (1,Floor) channel annotation, instances of *TemperatureSampler* for floor i are connected only to instances of *AverageTemperatureCalculator* destined for floors $i-1$, i, and $i+1$.

Formally, the ITaG $I(T,IC)$ is a graph whose vertices are in a set T of instantiated tasks and whose edges are from the set IC of instantiated channels. For each task AT_i in the abstract task graph from which I is instantiated, there are $f(AT_i,N)$ elements in T, where f maps the abstract task to the number of times it is instantiated in N. $IC \subseteq T \times T$ connects the instantiated version of the tasks. The ITaG I can also be represented as a graph $G(V,A)$, where $V = T$ and $A = IC$. Additionally, each T_j in the ITaG has a label indicating the subset of nodes in N it is to be deployed on. This overlay of communicating tasks over the target deployment enables the use of modified versions of classical techniques meant for analysing task graphs.

For example, for the application in Figure 2, since there are seven nodes with attached temperature sensors, $f(AT_1,N) = 7$, following the `every(AttachedSensors,TemperatureSensor` instantiation rule of the *TemperatureSampler* task. Similarly, $f(AT_2,N) = 3$, since the *AverageTemperatureCalculator* task is to be instantiated once on each of the three floors. The figure shows one allocation of the tasks in T, with arrows representing the instantiated channels in IC (it shows channels leading to only one instance of AT_2 for clarity). Note that the although the ITaG notation captures the information stored in the abstract task graph (including the instantiation rules of the tasks and the scopes of the connecting channels) it does not capture the *firing rules* associated with each task. The compiler's task involves incorporating the firing rule information while making decisions about allocating the tasks on the nodes.

Task Mapping: This task graph with composed channels is then instantiated on the given target network. Figure 2 illustrates an example of a target network. The nodes are on three different floors, and those marked with a thermometer have temperature sensors attached to them. In this stage, the compiler computes the mapping $M : T \rightarrow N$, while satisfying the placement constraints on the tasks.

Customisation of Runtime Modules: Based on the final mapping of tasks to nodes, and the composed channels, the *Datapool* and *ATaGManager* modules are configured for each node to handle the tasks and data items associated with it.

2.3 Changes to ATaG

For our work, we augment the above in two ways:

1. The abstract task graph now includes *end-to-end non functional requirements*. We use latency in this paper as an example, but our approach is applicable to other requirements as well.
2. The network description contains probability estimates of the time it takes for messages to be routes between nodes in same and neighbouring regions.

Using the above, we can reason about applications such as the Highway Traffic management application from [Mottola et al., 2007] (shown in Figure 3), and answer questions such as "Is it possible to deploy this application in a given city so that whenever the average speed of a highway sector goes to 0, the ramp signal for that sector turns red within 1 second with 98% probability?". The details of the above are discussed next.

3 PROBLEM DEFINITION

The task mapping problem with end-to-end requirements is solved in several steps. In Figure 4 we show a high-level description of each component involved in the process of mapping the tasks to the nodes, and the input and output of our constraint programming (CP) approach. As explained in programming model Section 2.1, the developer specifies an abstract task graph and the rules required to instantiate the tasks.

We preprocess the abstract task graph and instantiate tasks according to the rules specified in the task graph, which results in *instantiated* task graph and placement constraints, as shown in Figure 4, where the placement constraints enforce a task to be mapped only to a subset of sensor nodes.

The probability model, provided by the developer, specifies the delay distribution for sending a message between nodes in different and the same regions, as well as rules on how to combine probability distributions when messages travel more than one hop. This

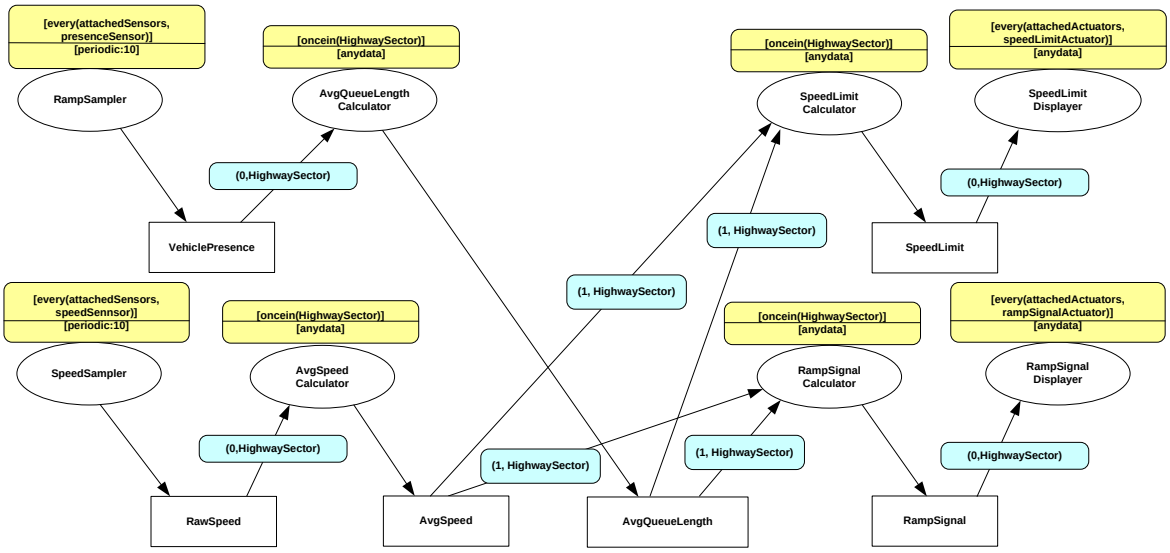

Figure 3: ATaG Program for the traffic app in [Mottola et al., 2007]

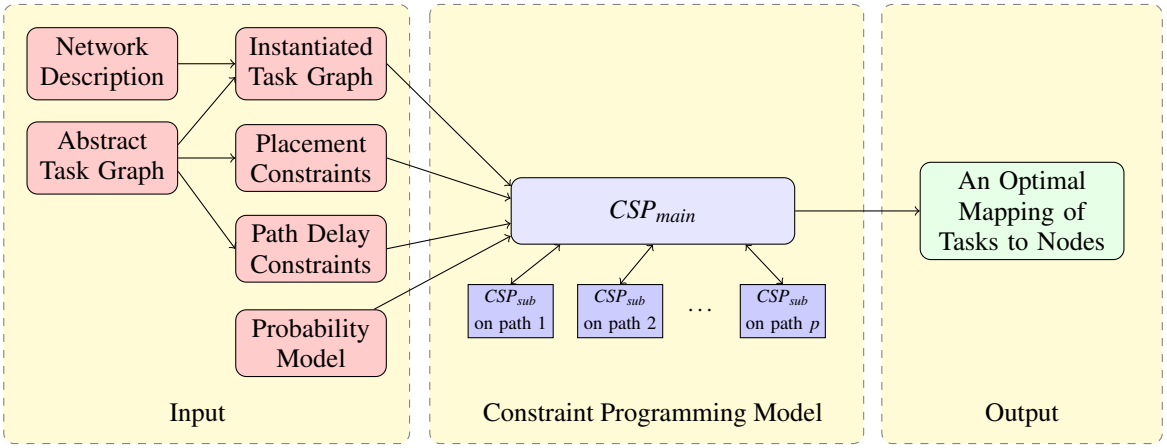

Figure 4: Block diagram representing the main components of the task mapping problem with end-to-end requirements.

allows both independent and dependent message delays to be modelled. In our experiments, for simplicity, we have assumed that all distributions are independent and normally distributed. In the mathematical formulation in Section 4, we formulate the aggregation of the delay requirements over the channels.

In our approach, we use a constraint programming model called CSP_{main} to perform the compilation process (see Section 2.2). The constraint programming model CSP_{main} takes the input (instantiated task graph, placement constraints, and the probability model), and outputs the optimal mapping from tasks to nodes. The end-to-end requirements are expressed in terms of maximum allowed delay between two task types. For example, we require that the delay for sending one unit of data between the *operational* tasks and the *actuating* tasks be within a certain threshold. We first convert the end-to-end requirements into a set of paths (namely path delay constraints), where each path includes all the tasks (subset of the instantiated tasks) between the two end-points. The probability model should specify how the probability distribution of the delay is computed along each path, given a mapping from task to nodes, also it should specify how the probability distribution of delay is computed if we replicate the tasks along each path.

In our CSP_{main} model, we create one CSP_{sub} model per path delay constraint, where the CSP_{sub} model solves and returns all the valid solutions (mapping of tasks to nodes) for one path delay constraint. The rules provided in the probability model are used in the CSP_{sub} model to check (in polynomial time) whether a mapping from the tasks to the nodes along one path is valid. If the requirements on a path can-

not be satisfied, then we allow the CSP_{sub} model to replicate the operational tasks on the path to improve the probability of receiving the data at the end-point. The valid solutions returned by CSP_{sub} for each path delay constraints are then composed in the CSP_{main} model to find an optimal mapping of the tasks to the nodes, minimising the number of replicated tasks. In the next section, we show how CSP_{main} and CSP_{sub} are modelled, the probability is computed as we replicate tasks, and how the overall problem is solved to optimality.

4 MATHEMATICAL FORMULATION

We first instantiate the abstract task on the target network, this process creates several copies of tasks per region. We then have the following constants in our model:

- Let N be the set of WSN nodes.
- Let T be the set of instantiated tasks.
- Let A be the set of arcs/edges in the directed acyclic task graph $G = (T, A)$, with edge (t_i, t_j) indicating that instantiated task t_i is sending data to task t_j.
- Let $D[t_i, t_j]$ be the random variable representing the delay for sending one unit of data from the task t_i to t_j, where $(t_i, t_j) \in A$ (the tasks are on the same edge of the task graph G).
- Let $F_D[t_i, t_j]$ be the cumulative distribution function (c.d.f) of the random variable $D[t_i, t_j]$.
- Let $S \subseteq T$ be the set of (start) tasks where a triggering event is produced.
- Let $E \subseteq T$ be the set of (end) tasks where producing an output within a given latency time is required.

For each task t let $node[t]$ be the decision variable denoting the node that task t is mapped to, with $t \in T$ and $node[t] \in N$.

As explained in Section 3, the task mapping problem with end-to-end requirements consists of *placement* and *path delay* constraints. The placement constraints enforce that a task may be restricted to a suitable subset of the sensor nodes, and therefore it cannot be mapped to any other node. Such placement constraints are modelled as follows:

$$node[t] \neq n \qquad \text{for every task } t \text{ that cannot be mapped to node } n \tag{1}$$

The path delay constraints ensure that for every path in the task graph G starting from the tasks in S and ending with the tasks in E the delay requirements are satisfied:

$$\forall\, t_i \in S,\ t_j \in E, \quad P\left(L[t_i, t_j] \leq maxDelay\right) \geq minProbability \tag{2}$$

where $L[t_i, t_j]$ is a random variable representing the latency for producing an output data by the task $t_i \in S$ in response to a triggering event produced by the task $t_j \in E$, and the constants *maxDelay* and *minProbability* are the requirements of the task graph expressed by the developer. The allowed maximum delay is represented with *maxDelay*, and *minProbability* is the minimum probability threshold for the delay requirement probability to hold, where delay is at most *maxDelay*. The path delay constraints (2) are equivalent to:

$$\forall\, t_i \in S,\ t_j \in E, \quad F_L[t_i, t_j]\left(maxDelay\right) \geq minProbability \tag{3}$$

where $F_L[t_i, t_j]$ is the cumulative distribution function (c.d.f) of the latency random variable $L[t_i, t_j]$. In our formulation, we consider $F_L[t_i, t_j]$ as an auxiliary variable, that should specify the c.d.f of the delay in the mapping of the tasks to the nodes on the path from t_i to t_j, and depends on $F_D[u, v]$ for each edge (u, v) in the path from t_i to t_j, and the rules specified by the probability model on how to combine the distributions over a path.

Typically, the tasks in the set S are operational tasks, e.g. *AverageSpeedCalculator*, and the tasks in the set E are the end-points of the data flow in the task graph, e.g. *RampSignalDisplayer*. For example, Figure 5 shows a path from the task $t_1 \in S$ of the type *AverageSpeedCalculator* to the task $t_3 \in E$ of the type *RampSignalDisplayer*, with the delay random variables $D[1,2]$ and $D[2,3]$ for the two edges on the path. In this example, as we consider the delay random variables $D[1,2]$ and $D[2,3]$ independent, it is trivial to see that the latency $L[t_1, t_3]$ is equal to the sum of the delay random variables on each edge:

$$L[t_1, t_3] = D[1,2] + D[2,3] \tag{4}$$

We observe that according to the path delay constraints (2) the probability $P(L[t_1, t_3] \leq 3.0) \geq 0.98$ states that for every event triggered by the *AverageSpeedCalculator* task t_1, we require that in 98% of

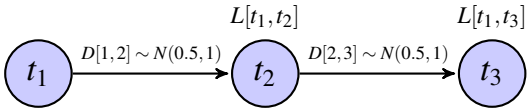

Figure 5: A path from the task t_1 to t_3 via task t_2 in the task graph, and the delay random variable $D[1,2]$ and $D[2,3]$ for the edges.

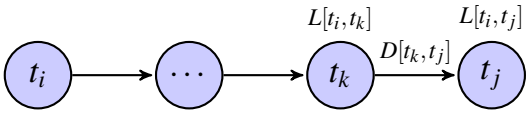

Figure 6: Path from the task t_i to t_j via task t_k in the task graph, and the delay random variable $D[t_k,t_j]$ for sending a unit of data from task t_k to t_j.

the time the *RampSignalDisplayer* task t_3 responds within 3.0 seconds (this corresponds to the requirement that in the event of a traffic jam, no more cars should be allowed onto the highway). If we assume the delay random variables $D[1,2]$ and $D[2,3]$ are normally distributed with mean 0.5 and variance 1 ($N(0.5,1)$), then the c.d.f of $L[t_1,t_3]$ is also a normal distribution with mean 1 and variance 2 (sum of the means and variances). Therefore, the probability $P(L[t_1,t_3] \leq 3.0) \geq 0.98$ becomes $F_L[t_1,t_3](3.0) = 0.92135 \not\geq 0.98$, which is not satisfiable. This implies that given the requirements $maxDelay = 3$ and $minProbability = 0.98$ and in the presence of delay on both of the channels $D[1,2]$ and $D[2,3]$, it is not possible to receive a message at the end-point task t_3, within 3 seconds in at least 98% of the times.

Figure 6 shows an arbitrary path in the task graph starting from the task $t_i \in S$ and ends at $t_j \in E$. The latency $L[t_i,t_j]$ at the task t_j is the sum of the delay for all edges in the path originated from the task t_i:

$$L[t_i,t_j] = L[t_i,t_k] \oplus D[t_k,t_j] \tag{5}$$

where $L[t_i,t_k]$ is the latency of the direct predecessor of the task t_j namely t_k in Figure 6, and $\oplus$ is the rule provided by the probability model (see Section 3) for combining the delay distribution over channels. We consider that there is only one path from t_i to t_j in the task graph, and from here throughout the paper, we assume that the delay distribution on all channels are independent, hence the operator $\oplus$ can be simplified the sum operator $+$.

Computing the distribution of the random variable latency $L[t_i,t_j]$ based on the task mapping variables $node[t]$ is not trivial. In order to simplify the process, we first show the connection between the mapping variables $node[t]$ to the delay distribution of each edge of the task graph G.

We consider that the delay for transmitting a message between two tasks placed on the same sensor node is 0. Therefore, the random variable $D[t_i,t_j]$ becomes 0 if $node[t_i] = node[t_j]$ (tasks t_i and t_j are mapped to the same node), and the c.d.f $F_D[t_i,t_j]$ becomes the *degenerate* distribution $F(D;0)$ [Karr, 1993]:

$$F_D[t_i,t_j] = \begin{cases} F(D;0) & \text{if } node[t_i] = node[t_j] \\ \Phi_{\mu_{i,j},\sigma_{i,j}} & \text{if } node[t_i] \neq node[t_j] \end{cases} \tag{6}$$

where $\Phi_{\mu_{i,j},\sigma_{i,j}}$ is the c.d.f of the normal distribution with the mean $\mu_{i,j}$ and the standard deviation $\sigma_{i,j}$. The c.d.f $F_D[t_i,t_j]$ of the delay for all tasks in the same region is the same, e.g. normal distribution. However, $F_D[t_i,t_j]$ can be a different distribution for tasks transmitting data between different regions. In this work, to simplify the presentation, we consider $F_D[t_i,t_j] = \Phi_{\mu,\sigma}$ for all tasks, but the parameters μ and σ might differ from region to region. Given the delay distribution on the edges (6) and the operator $\oplus$ in (5), we can compute the c.d.f of L. For example, let the delay distribution for the edges in the path shown in Figure 5 be the same as in (6). The c.d.f of the delay on the path from the task t_1 to t_3 shown in Figure 5 is:

$$F_L[t_1,t_3] = \begin{cases} F(D;0) & \text{if } node[t_1] = node[t_2] \wedge \\ & \quad node[t_2] = node[t_3] \\ \Phi_{\mu_{1,2},\sigma_{1,2}} & \text{if } node[t_1] \neq node[t_2] \wedge \\ & \quad node[t_2] = node[t_3] \\ \Phi_{\mu_{2,3},\sigma_{2,3}} & \text{if } node[t_1] = node[t_2] \wedge \\ & \quad node[t_2] \neq node[t_3] \\ \Phi_{\mu_{1,2}+\mu_{2,3},\sigma_{1,2}+\sigma_{2,3}} & \text{if } node[t_1] \neq node[t_2] \wedge \\ & \quad node[t_2] \neq node[t_3] \end{cases} \tag{7}$$

If all the tasks are placed on the same sensor node (the first case in (7)) the c.d.f of two degenerate distribution with parameter 0 is also a degenerate distribution with parameter 0. In the second and the third case in (7), one pair of the nodes are placed on the same sensor node and the other pair are not, and the c.d.f of the sum of a degenerate distribution with parameter 0 and a normal distribution is still a normal distribution maintaining the same parameters. Finally, in the last case of (7), if all the tasks are placed on different sensor nodes, then the c.d.f of the sum of the two random variables $D[t_1,t_2]$ and $D[t_2,t_3]$ with normal distribution is also normal distribution with its mean being the sum of the two means, and its variance being the sum of the two variances ($\Phi_{\mu_{1,2}+\mu_{2,3},\sigma_{1,2}+\sigma_{2,3}}$), as we assume the delays are independent.

As shown in equation (7), for three *adjacent* pairs of tasks on the path from t_1 to t_3, choosing ($node[t_i] = node[t_j]$) or ($node[t_i] \neq node[t_j]$), where $t_i,t_j \in \{t_1,t_2,t_3\}$, creates $2^{3-1} = 4$ possible combinations, and as a result $F_L[t_1,t_3]$ is a piecewise function with four sub-function. In the general case, for k *adjacent* pairs of tasks on the path from t_i to t_j, choosing ($node[t_i] = node[t_j]$) or ($node[t_i] \neq node[t_j]$) creates 2^{k-1} possible combinations, hence, the c.d.f of the delay random variable $L[t_i,t_j]$ is a piecewise function with 2^{k-1} sub-functions.

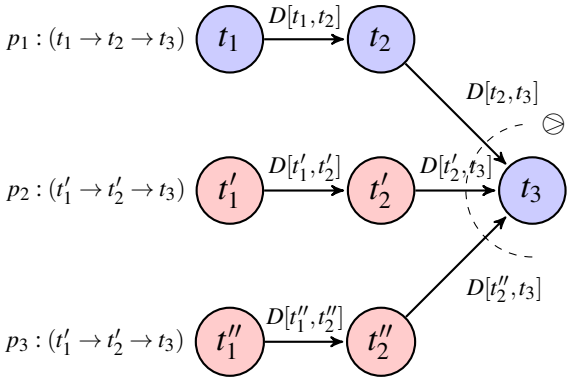

Figure 7: A path from the task t_1 to t_3 via task t_2 in the task graph, the two copies of the tasks t_1 and t_2 denoted by t'_1, t''_1 and t'_2, t''_2 respectively, and the delay random variable $D[t_i,t_j]$ for an edge $(t_i,t_j) \in A$.

4.1 TASK REPLICATION

The path delay constraints (3) depends on the constants *maxDelay* and *minProbability*, which is provided by the developer as part of the task graph. Therefore, it is possible that these constraints are not satisfiable. To tackle this issue, we replicate certain tasks more than once in a region, for the express purpose of reducing the possible delay values of the random variables $L[t_i,t_j]$ (since t_j can fire as soon as one of the clones' data is available). This will incur a cost in terms of communication overhead. However, we formulate the problem to minimise the number of replicated tasks and hence minimise the communication overhead.

In Figure 7, we show an example of task replication on the path p_1 from the task t_1 to t_3, with the tasks t'_1, t''_1 replicated from the task t_1, and the tasks t'_2, t''_2 replicated from the task t_2. A replicated task also replicates the incoming and outgoing edges from the direct predecessor and to the direct successor tasks in the task graph. We only replicate the operational tasks (namely t_1 and t_2 in Figure 7), and as a result the two new paths p_2 and p_3 share the actuator task t_3.

The latency for the delay at an end-point t_3 on the path p_1 is equal to the minimum of the delay random variable L of all paths p_1, p_2 and p_3:

$$L[t_1,t_3] = \min\left(L[t_1,t_3],\ L[t'_1,t_3],\ L[t''_1,t_3]\right)$$

In the general case, the delay random variable $L[t_i,t_j]$ along a path with the tasks replication is computed by an additional operator $\circledcirc$ (operator min in our work, which is provided by the probability model) to compose the properties of the channels over the (parallel) fan-in at the end-point task t_j:

$$L[t_i,t_j] = \mathop{\circledcirc}_{(t_k,t_j)\ \in\ A'_j} \left(L[t_i,t_k] \oplus D[t_i,t_j]\right) \tag{8}$$

where A'_j is the set of all replicated edges for the direct predecessor of the task t_j including the non-replicated edge on the path from t_i to t_j.

Based on order statistics [Arnold et al., 2008, David and Nagaraja, 2003], the c.d.f of $L[t_i,t_j]$ in (8) with the operator $\circledcirc$ being the minimum of independent random variables becomes:

$$\forall\ t_i \in S,\ t_j \in E,$$
$$F_L[t_1,t_3] =$$
$$1-(1-F_L[t_i^1,t_j])\cdot(1-F_L[t_i^2,t_j])\cdots(1-F_L[t_i^r,t_j]) \tag{9}$$

where t_i^r represents the r^{th} copy of the task t_i.

For example, in Figure 7, we can use the equation (9) to compute the latency at the task t_3. Assume on the path p_1 the tasks can only be placed on different nodes ($node[t_1] \neq node[t_2] \neq node[t_3]$), also let $maxDelay = 2$, $minProbability = 0.98$, and the delay distribution be normal distribution $N(0.5,1)$ for all edges. From (7), before replicating the tasks, the c.d.f $F_L[t_1,t_3]$ of the delay random variable on path p_1 becomes:

$$node[t_1] \neq node[t_2] \wedge node[t_2] \neq node[t_3] \implies$$
$$F_L[t_1,t_3] = \Phi_{0.5+0.5,1+1} = \Phi_{1,2}$$

and the probability of the latency on the path p_1 being at most 2.0 seconds is $F_L[t_1,t_3](2) = 0.76025$. However, $0.76025 \not\geq 0.98$ and the constraint (3) is not satisfiable. In order to satisfy constraint (3), we create two copies of the tasks t_1 and t_2 as shown in Figure 7. The replicated tasks maintain the same properties and the originated tasks. Hence, the probability of latency being at most 2 seconds on path p_2 and p_3 is also 0.76025. Using equation (9) we have:

$$F_L[t_1,t_3] =$$
$$1-(1-0.76025)\cdot(1-0.76025)\cdot(1-0.76025) = 0.9862$$

and the constraint (3) is satisfied ($0.9862 \geq 0.98$).

The task mapping optimisation problem with tasks replication becomes:

$$\text{minimise } |T'|$$

subject to:

$$node[t] \neq n \qquad \begin{array}{l}\text{for every task } t \text{ that}\\ \text{cannot be mapped to node } n\end{array} \tag{10}$$

$$\forall\ t_i \in S,\ t_j \in E,$$
$$F_L[t_i,t_j]\,(maxDelay) \geq minProbability \tag{11}$$

where T' is the set of all replicated tasks, and $F_L[t_i,t_j]$ is computed from (9).

4.2 TASK REPLICATION LOWER BOUND

For any given path p from the task t_i to the task t_j, the mapping of all tasks to different nodes incurs the maximum latency at the end-point task t_j. Let MF be the maximum latency incurred by assigning all tasks to different nodes ($MF = \max(F_L[t_i,t_j])$), and r be the number of replicated tasks. The lower bound on the number of tasks replicated is:

$$\begin{aligned} F_L[t_i,t_j] = 1-(1-MF)^r \geq minProbability &\implies \\ (1-MF)^r \leq 1-minProbability &\implies \\ r\cdot\log(1-MF) \leq \log(1-minProbability) &\implies \\ r \geq \frac{\log(1-minProbability)}{\log(1-MF)} & \end{aligned} \tag{12}$$

For example, as we have shown in Figure 7 with $minProbability = 0.98$ and $MF = 0.76025$, we require to replicate the tasks on the path p_1 at least two times, which can also be derived using the lower bound (12):

$$r \geq \frac{\log(1-0.98)}{\log(1-0.76025)} = 2.74121$$

In our implementation, we use the floor of the lower bound in (12) to compute the minimum number of replicated tasks required.

5 CONSTRAINT PROGRAMMING APPROACH FOR END-TO-END REQUIREMENTS

As we have seen in Section 4, the abstract nature of the path delay constraints (3) and the c.d.f of latency (7) at an end-point being a piecewise function of the decision variables $node[t]$, make it very difficult to implement and solve the task mapping optimisation problem with end-to-end requirements using conventional approaches and optimisation solvers. In this section, we use constraint programming (CP) to implement the mathematical model stated in Section 4.

Algorithm 1 lists our CP approach in solving the task mapping problem with delay requirements. In our main CP model (CSP_{main} in line 2), each path delay constraint (3) is modelled as a sub constraint satisfaction problem (CSP_{sub} in line 8), and the whole solution to each sub-CSP is expressed as an *extensional* constraint (also known as user-defined or ad-hoc constraint). Extensional constraints provide a way to specify all valid solutions to the constraint, using either a deterministic finite automaton or a tuple set (*table* constraint). In this work, we use a tuple set to express all the valid solutions to the path delay constraints (3) and enforce these constraints in our main CP model.

Algorithm 1: The task mapping with end-to-end delay requirements algorithm

input : N, T, S, E, *minProbability*
output: *node*

```
1  solved ← false
2  add placement constraints to CSP_main
3  while not solved do
4      taskCopiesRequired ← false
5      forall t_i ∈ S, t_j ∈ E do
6          tupleSet[t_i,t_j] ← ∅
7          taskCopies ← 0
8          forall solutions s in CSP_sub⟨p[t_i,t_j]⟩ do
9              if checker(s, MF) then
10                 tupleSet[t_i,t_j] =
                   tupleSet[t_i,t_j] ∪ s
11         if tupleSet[t_i,t_j] ≠ ∅ then
12             add extentsional constraints to
               CSP_main
13         else
14             taskCopies ← log(1−minProbability)/log(1−MF)
15             replicate(p[t_i,t_j], taskCopies)
16             taskCopiesRequired ← true
17     if not taskCopiesRequired then
18         solve(CSP_main)
19         if CSP_main has a solution then
20             solved ← true
21             return node
22         else
23             replicate (1)
```

Our CSP_{main} model enforces the placement constraints (1) and the following extensional constraint representing the valid solutions to the path delay constraints (3):

$$\begin{aligned} &\forall\, t_i \in S, t_j \in E \\ &extensional\,(p[t_i,t_j],\ tupleSet[t_i,t_j]) \end{aligned} \tag{13}$$

where $p[t_i,t_j]$ is the set of all tasks on the path from the task t_i to the task t_j including t_i and t_j, and $tupleSet[t_i,t_j]$ is the set of tuples, where each tuple is a valid solution to the path delay constraint (3), given by solving the CSP_{sub} (line 8). The extensional constraints (13) are then constraining the valid solutions of the path delay constraints to the corresponding variables $node[t],\ t \in p[t_i,t_j]$ (line 12), on which

domain consistency (all values not participating in a solution are pruned) is achieved.

For example, in Figure 5, let the delay random variable D be a normal distribution $N(0.5,1)$, and the domain of the tasks be as follows:

$$node[t_1] = \{1,2\},\ node[t_2] = \{1,2\},\ node[t_3] = \{2\}$$

also let $maxDelay = 3$, $minProbability = 0.98$. The path delay constraints according to (7) become:

$$F_L[t_1,t_3] = \begin{cases} F(D;0) & \text{if } node[t_1,t_2,t_3] = [3,3,3] \\ \Phi_{0.5,1} & \text{if } node[t_1,t_2,t_3] = [2,3,3] \\ \Phi_{0.5,1} & \text{if } node[t_1,t_2,t_3] = [2,2,3] \\ \Phi_{1,2} & \text{if } node[t_1,t_2,t_3] = [3,2,3] \end{cases} \tag{14}$$

where $node[t_1,t_2,t_3] = [3,3,3]$ is a vector notation for the assignment $node[t_1] = 3$, $node[t_2] = 3$, $node[t_3] = 3$. The path delay constraints (3) for all the possible assignments in (14) become:

$$node[t_1,t_2,t_3] = [3,3,3] \implies F_L[t_1,t_3](3) = F(D;0) = 1 \geq 0.98 \tag{15}$$

$$node[t_1,t_2,t_3] = [2,3,3] \implies F_L[t_1,t_3](3) = \Phi_{0.5,1}(3) = 0.99379 \geq 0.98 \tag{16}$$

$$node[t_1,t_2,t_3] = [2,2,3] \implies F_L[t_1,t_3](3) = \Phi_{0.5,1}(3) = 0.99379 \geq 0.98 \tag{17}$$

$$node[t_1,t_2,t_3] = [3,2,3] \implies F_L[t_1,t_3](3) = \Phi_{1,2}(3) = 0.92135 \ngeq 0.98 \tag{18}$$

The possible assignments (15), (16), and (17) satisfy the path delay constraint, except the assignment $node[t_1,\ t_2,\ t_3] = [3,2,3]$ (18). Note that the $tupleSet[t_1,t_3]$ only includes valid assignments:

$$tupleSet[t_1,t_3] = \{[3,3,3],[2,3,3],[2,2,3]\}$$

and therefore in this example the extensional constraint (13) is:

$$extensional(\{t_1,t_2,t_3\},\ \{[3,3,3],[2,3,3],[2,2,3]\}) \tag{19}$$

The extensional constraint (19) constrains the decision variables $node[t_1]$, $node[t_2]$, and $node[t_3]$ to the given valid assignments by the *tupleSet*. In the presence of many extensional constraints for several path delay constraints, our CSP_{main} model can effectively prune the values that do not participate in a solution, using the produced valid assignments *tupleSet* given by the CSP_{sub} model.

The CSP_{sub} model produces solutions to the ground instance of the task mapping problem with only decision variables limited to one path on the task graph, and only includes placement constraints. Therefore, it is efficient to produce all solutions and check the validity of each solution using a `checker` function (line 9). The `checker` function computes the c.d.f of the delay according to (9) and incrementally maintains the maximum value of the c.d.f (MF) to be used in computing the lower bound *taskCopies* on the tasks replication (line 14). The `checker` function returns true if in the assignment s the c.d.f of the delay is at least *minProbability* satisfying the path delay constant (3), and false otherwise. If the `checker` function returns true, the valid solution s is added to the $tupleSet[t_i,t_j]$ (line 10), and if there is at least one valid solution ($tupleSet[t_i,t_j] \neq \emptyset$, line 11), then the extensional constraint (13) is added to the CSP_{main} model (line 12). Note that the initial domain reduction is performed by the extensional constraints at this point (all values not participating in a solution to the CSP_{main} are pruned from the domains of the decision variables), which reduces the number of solutions for the consecutive CSP_{sub} models.

If there is no valid solution to CSP_{sub} ($tupleSet[t_i,t_j] = \emptyset$, line 13), then at least one of the path delay constraints can not be satisfied, hence CSP_{main} is also not satisfiable, and we required to replicate the tasks. We replicate the tasks such that the path delay constraints over the path p are satisfied, and we start from the mathematical lower bound on the number of task copies *taskCopies* (line 14). The `replicate` function (line 15) takes the path $p[t_i,t_j]$ and creates *taskCopies* replicates of the operational tasks along the path maintaining the same incoming and outgoing edges in the path $p[t_i,t_j]$ from the task graph.

We solve CSP_{main} (line 20) only if all path delay constraints are individually satisfiable (no task replicate per path basis is required, $taskCopiesRequired =$ **false**, line 17). If CSP_{main} has a solution, the *node* decision variables are returned (line 21) and the algorithm ends, otherwise CSP_{main} became unsatisfiable due to the composition of all path delay constraints, therefore we add one more copy of each operational tasks in the task graph G (line 23).

We further optimise the CSP_{sub} model by adding symmetry breaking constraints [Rossi et al., 2006]. Let $p_k[t_i,t_j]$ be the k^{th} set of replicated tasks on the path from t_i to t_j, where $p_0[t_i,t_j]$ represents the initial tasks from t_i to t_j. We refer to $p_k[t_i,t_j]$ as the k^{th} replicated path. Let $v_k[t_i,t_j]$ be the vector of the decision variables $node[t]$ on the k^{th} replicated path in the order of the directed edges. We enforce lexicographic ordering to remove the equivalent permutation of val-

ues in a solution to CSP_{sub} between replicated paths, where there is at least one replicated path required:

$$\forall p_k[t_i,t_j],\ 0 \leq k \leq NR[t_i,t_j] \\ v_k[t_i,t_j] \preceq_{lex} v_{k+1}[t_i,t_j] \tag{20}$$

where $NR[t_i,t_j]$ is the number of replicated paths from task t_i to t_j and $\preceq_{lex}$ is the lexicographic ordering constraint between the vectors of decision variables $v_k[t_i,t_j]$ and $v_{k+1}[t_i,t_j]$.

6 EVALUATION

6.1 Experimental Setup

In our experiments, we use the realistic road traffic application and its latency requirements in [US Department of Transportation, Federal Highway Administration, 2012]. We start from the smallest highway traffic size of $\langle |N|,|T| \rangle = \langle 7,9 \rangle$ and increase the instance size by one region per instance (each region is a sector in a highway with at least 6 sensor nodes) up to the largest instance of highway traffic with 25 regions (highway traffic $\langle 150,216 \rangle$). We assume that the delay distribution for sending a message between any two regions is a normal distribution $N(0.5,1)$ with mean 0.5 and variance 1. For each instance, we enforce the end-to-end delay constraints between the *AverageSpeedCalculator* tasks and the *RampSignalDisplayer* tasks with the message delay probability of 0.98 within 1, 2, or 3 seconds.

Our CP model is implemented in *Gecode* [Gecode Team, 2006] (revision 4.2.0)[1] and runs under Mac OS X 10.8.4 64 bit on an Intel Core i5 2.6 GHz with 3MB L2 cache and 8GB RAM. We separately solve each instance for the maximum delay value (*maxDelay*) of 1, 2, or 3 seconds. We set a timeout of 60 seconds for each instance, recording the time to solve optimally and the number of replicated tasks.

6.2 Analysis of Results

In Figure 8, we present the average runtime results (in seconds) of 10 runs for solving the highway traffic instances with our CP model. Each instance is solved upon varying the delay requirement (*maxDelay*) on the values 1, 2, and 3 seconds. The lower values of *maxDelay* makes the instances more difficult to solve, as the delay requirement becomes harder to satisfy and more task replicates are required. All instances were solved optimally, minimising the number of replicated tasks.

[1]available from `http://www.gecode.org/`

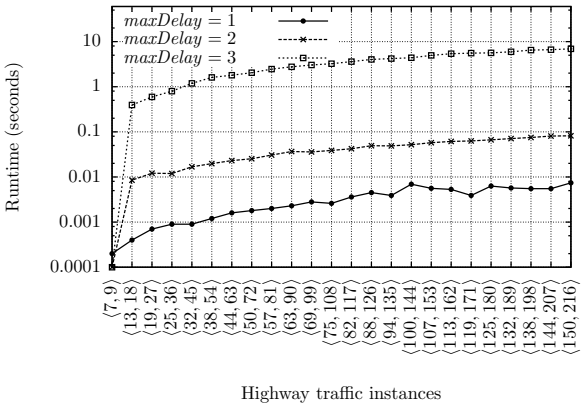

Figure 8: Average runtime in seconds of 10 runs and in logarithmic scale for solving optimally the task mapping problem with end-to-end requirements on the highway traffic application with a maximum allowed delay of 1, 2, or 3 seconds. The instances are represented with $\langle |N|,|T| \rangle$, where $|N|$ is the number of sensor nodes and $|T|$ is the number of instantiated tasks.

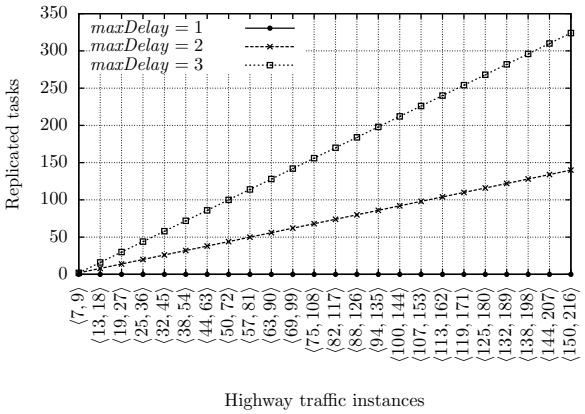

Figure 9: Number of replicated tasks required to satisfy the end-to-end requirements on the highway traffic application with a maximum allowed delay of 1, 2, or 3 seconds. The instances are represented with $\langle |N|,|T| \rangle$, where $|N|$ is the number of sensor nodes and $|T|$ is the number of instantiated tasks.

The runtimes for instances with $maxDelay = 2$ and 3 seconds are very short, therefore we take the average runtime of 10 runs. In Figure 8, we present the runtimes with logarithmic scale to further distinguish the difference of runtime between the instances with $maxDelay = 2$ and instances with $maxDelay = 3$ seconds. The runtime includes the preprocessing time to create and solve the CSP_{sub} models, and the runtime to solve the CSP_{main} model. The runtime increases linearly as the size of the problem increases and the maximum allowed delay (*maxDelay*) decreases. For

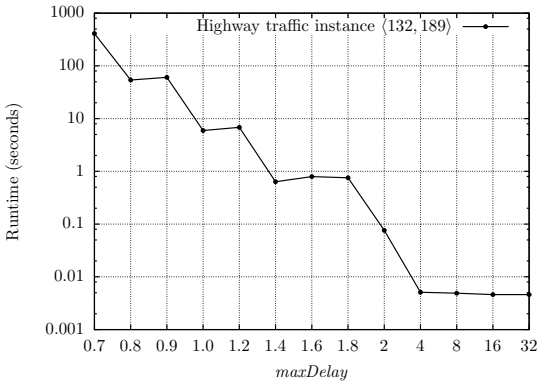

Figure 10: The average runtime in seconds and logarithmic scale for solving highway traffic instance $\langle 132, 189 \rangle$ with 10 runs, varying *maxDelay*.

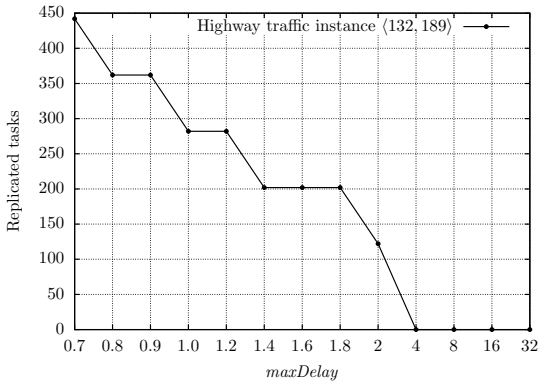

Figure 11: Number of replicated tasks required to satisfy the end-to-end requirements on the highway traffic instance $\langle 132, 189 \rangle$, varying *maxDelay*.

the first instance $\langle 7, 9 \rangle$, the runtime is significantly shorter than larger instances, because this instance involves only one region with one path delay constraint, and requires only 2 task replicates to satisfy the delay requirement when $maxDelay = 1$ or 2 seconds. The runtime drastically increases as the maximum allowed delay drops below the average delay on a path between two end-points with delay requirements. In our instances, the paths between the two end-points includes three edges (path length is 3), where the average delay on each edge is 0.5 second, therefore the average delay between the two end-points on the path is 1.5 seconds, and choosing $maxDelay = 1$ second requires several iterations of creating task replicates, hence the longer runtimes. The standard deviation of the runtimes for all instances is on average 0.05 seconds, and at most 1.69 seconds, while varying *maxDelay* between 1 and 3 seconds with 10 runs.

In Figure 9 we present the number of replicated tasks in order to satisfy the end-to-end requirements expressed by the path delay constraints in our CP model. As we expected, a tighter requirement on the allowed maximum delay (*maxDelay*) requires more replicated tasks to improve the chance of a message delivery at an end-point and to satisfy the path delay constraints. As the delay requirement *maxDelay* drops from 2 seconds to 1 second, the replicated tasks are not exactly doubled, as some path delay constraints are satisfied with fewer replicated tasks. However, increasing the instance size by one region increases the number of task replicates equally between two consecutive instances, which is due to the equal delay distribution between all regions.

In Figure 10, we present the average runtime results (in seconds) of 10 runs for solving a reasonably large instance of the highway traffic application (namely instance $\langle 132, 189 \rangle$), varying the values of *maxDelay* from 0.7 to 32 seconds. In Figure 11, we show the results for the number of replicated tasks with the same setup as in Figure 10. The result shows that the runtime and the number of replicated tasks follow the same trend, while varying the maximum allowed delay *maxDelay*. The runtime tightly depends on the number of replicated tasks, and there is an interval on the values of *maxDelay*, where the average runtime and the number of replicated tasks does not change significantly.

These results show that constraint programming can efficiently solve the combinatorial problem with end-to-end requirements introduced in task mapping for WSNs.

7 CONCLUSION

In this paper, we formulated the problem as a constraint program of task-mapping while honouring end-to-end requirements encountered during sensor network macroprogramming, and presented efficient algorithms to solve it. We also addressed the case when copies of certain tasks are permitted in order to increase performance guarantees. Our evaluations, performed on a realistic highway traffic application task graph for the special case of managing end-to-end latency, show that this problem can indeed be solved efficiently using our approach, although increased computation time is needed for tighter bounds. We investigated the specific case of latency requirement in this paper. However the end-to-end requirements are only given as an example. It is possible to address other end-to-end requirements by changing the rules of the probability model for combining the probability distributions along the path between the two end points. For example, if instead of latency we consider *link quality* as a random variable

over the channels, then the probability model must state the distribution of the random variable link quality and how it is combined over channels (path between two end-points). We then replace the operator $\oplus$ and $\ominus$ (8) in the the probability model with the operator *product* and the operator *min*, respectively, and our formulation (see Section 4) is still valid for maintaining the link quality requirements. Our immediate future work is in two parallel directions: *i)* integration of our approach into the publicly available Srijan toolkit[2] for ATaG, and *ii)* further exploring the overhead induced by creating copies of tasks and developing more accurate CP models to minimise it while achieving the desired non-functional guarantees. We also envision the application of our work in cloud computing and related technologies, where guaranteeing certain requirements on the services running in the cloud is essential, and latencies among co-located nodes are similar to those in different data centres.

ACKNOWLEDGEMENTS

This research is partially sypported by the Swedish Foundation for Strategic Research (SSF) under research grant RIT08-0065 for the project *ProFuN*, and the French ANR-BLAN-SIMI10-LS-100618-6-01 MURPHY project. Special thanks to the reviewers for their comments.

REFERENCES

Arnold, B. C., Balakrishnan, N., and Nagaraja, H. N. (2008). *A first course in order statistics*, volume 54 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA. Unabridged republication of the 1992 original.

Bakshi, A., Pathak, A., and Prasanna, V. K. (2005a). System-level support for macroprogramming of networked sensing applications. In *Intl. Conf. on Pervasive Systems and Computing (PSC)*.

Bakshi, A., Prasanna, V. K., Reich, J., and Larner, D. (2005b). The Abstract Task Graph: A methodology for architecture-independent programming of networked sensor systems. In *Workshop on End-to-end Sense-and-respond Systems (EESR)*.

Choi, W., Shah, P., and Das, S. (2004). A framework for energy-saving data gathering using two-phase clustering in wireless sensor networks. In *Proc. of the 1st Int. Conf. on Mobile and Ubiquitous Systems: Networking and Services (MOBIQUITOUS)*.

David, H. A. and Nagaraja, H. N. (2003). *Order statistics*. Wiley Series in Probability and Statistics. Wiley-Interscience [John Wiley & Sons], Hoboken, NJ, third edition.

Gecode Team (2006). Gecode: A generic constraint development environment. `http://www.gecode.org/`.

[2]http://code.google.com/p/srijan-toolkit/

Gummadi, R., Gnawali, O., and Govindan, R. (2005). Macro-programming wireless sensor networks using Kairos. In *Proc. of the 1st Int. Conf. on Distributed Computing in Sensor Systems (DCOSS)*.

Hassani Bijarbooneh, F., Flener, P., Ngai, E., and Pearson, J. (2011). Energy-efficient task mapping for data-driven sensor network macroprogramming using constraint programming. In *Operations Research, Computing, and Homeland Defense*, pages 199–209. Institute for Operations Research and the Management Sciences.

Karr, A. (1993). *Probability*. Springer Texts in Statistics Series. Springer-Verlag.

Mottola, L., Pathak, A., Bakshi, A., Prasanna, V. K., and Picco, G. P. (2007). Enabling scope-based interactions in sensor network macroprogramming. In *Proc. of the 4th Int. Conf. on Mobile Ad-Hoc and Sensor Systems*.

Mottola, L. and Picco, G. (2011). Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3):19.

Newton, R., Morrisett, G., and Welsh, M. (2007). The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498. ACM.

Pathak, A., Mottola, L., Bakshi, A., Prasanna, V. K., and Picco, G. P. (2007). Expressing sensor network interaction patterns using data-driven macroprogramming. In *Third IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing (PerSeNS 2007)*.

Pathak, A. and Prasanna, V. K. (2010). Energy-efficient task mapping for data-driven sensor network macroprogramming. *IEEE Transactions on Computers*, 59(7):955–968.

Pathak, A. and Prasanna, V. K. (2011). High-Level Application Development for Sensor Networks: Data-Driven Approach. In Nikoletseas, S. and Rolim, J. D., editors, *Theoretical Aspects of Distributed Computing in Sensor Networks*, Monographs in Theoretical Computer Science. An EATCS Series, pages 865–891. Springer Berlin Heidelberg.

Rossi, F., van Beek, P., and Walsh, T., editors (2006). *Handbook of Constraint Programming*. Elsevier.

Tian, Y., Ekici, E., and Özgüner, F. (2005). Energy-constrained task mapping and scheduling in wireless sensor networks. In *IEEE International Conference on Mobile Ad hoc and Sensor Systems*, pages 8–218. IEEE Computer Society Press.

US Department of Transportation, Federal Highway Administration (2012). Manual on uniform traffic control devices: Preemption and priority control of traffic control signals. `http://mutcd.fhwa.dot.gov/pdfs/2009r1r2/part4.pdf`. Section 4D.27.

Wu, Y., Kapitanova, K., Li, J., Stankovic, J. A., Son, S. H., and Whitehouse, K. (2010). Run time assurance of application-level requirements in wireless sensor networks. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, IPSN '10.