SUMO User Conference 2022
https://doi.org/10.52825/scp.v3i.123

Published: 29 Sept. 2022

Implementation of a Perception Module for Smart Mobility Applications in Eclipse MOSAIC

Robert Protzmann[1], Karl Schrab[2], Moritz Schweppenhäuser[1], and Ilja Radusch[2]

[1]Fraunhofer Institute FOKUS, Berlin, Germany
{robert.protzmann, moritz.schweppenhaeuser}@fokus.fraunhofer.de
[2] Daimler Center for Automotive IT Innovations (DCAITI),
Technical University of Berlin, Germany
{karl.schrab, ilja.radusch}@dcaiti.com

Abstract

Nowadays, smart mobility applications could benefit from environment perception, en-abled by evolving sensor technology and processing capabilities available for traffic entities. On the application level, in many cases, information about detected objects is required in-stead of the raw sensor data. Developing and evaluating the impacts of such applications can be done in co-simulation frameworks, which combine the modeling of different domains such as application, communication, and traffic. Eclipse MOSAIC is a suitable solution for this task, combining the traffic simulation of Eclipse SUMO with other simulators, such as the integrated Application Simulator, or OMNeT++ and ns-3 for modeling commu-nication. However, a model for perceiving surrounding traffic entities, such as vehicles, traffic signals, and traffic signs, is only available to a limited extent. In this paper, we introduce an object-level perception module to the MOSAIC Application simulator. It takes advantage of state-of-the-art spatial indexing methods to get rapid access to traffic objects, especially moving objects, within a defined field of view. We furthermore evaluate the computational performance of the indexing techniques as well as the integration with the traffic simulator SUMO using *TraCI* and *Libsumo*. With the aid of this model, novel connected applications that analyze or share surrounding objects, e.g. for an improved traffic state estimation, can now be evaluated with Eclipse MOSAIC.

1 Introduction

Safety, efficiency, sustainability - connected vehicles and associated applications are promised to improve future mobility for more than a decade now. In order to evaluate the impact of these applications by simulation, we developed the framework Eclipse MOSAIC (formerly VSimRTI), which combines simulators from the different domains of applications, traffic, communication, and others, needed for holistic modeling of Intelligent Transport Systems (ITS) [4, 11].

From the view of the distributed applications, especially the MOSAIC Application simulator plays an essential role as it covers the most important participants in a traffic scenario. For this purpose, it models several entities, which are connected, and which are equipped with an application logic. First of all, *Vehicles* are considered as moving entities, whereas their movements and states are simulated in a vehicle simulator like PHABMACS [10] or Carla [2], or a traffic simulator like Eclipse SUMO [9] and synchronized to the Application Simulator. Due to the bidirectional coupling, applications in the Application Simulator could control probable state changes, which then influence the behavior in the vehicle or traffic simulator. *Pedestrians* are another kind of moving entity. As stationary entities, *Traffic Signals* are usually also directly related to the vehicle and traffic simulators, due to their close and also pre-defined interaction with the moving participants such as vehicles. Additional stationary entities are *Charging Stations*, *Road Side Units (RSUs)* and (novel) *Traffic Signs*. These entities might be part of physical infrastructure on or beside streets, but are not necessarily related to the vehicle and traffic simulator since their interaction might be part of the conducted research and require a dedicated implementation. *Cloud/Edge Servers* and *Traffic Management Centers* (which are specialized Servers) are also stationary, but usually not part of road infrastructure. They realize interaction with other entities by communication.

With evolving sensor technology and processing capabilities, more and more data are handled by the traffic entities, enabling novel mobility applications. Nowadays, radar, cameras, and even LiDAR have found a permanent place in vehicles for the detection of traffic objects in the surrounding area of the vehicle. The perceived information supports safety applications, but also efficiency could benefit. One example is estimating the traffic state on the road by considering surrounding vehicles, e.g., estimating the density on roads. Furthermore, parking space solutions could be imagined with direct detection by vehicles. Not only vehicles could be equipped with the perception technology, but also stationary entities like traffic signals or RSUs at certain intersections, or gantries on highways. All in all, there are many constellations of how traffic entities could perceive each other. For the simulation system, perception implies new interaction possibilities between entities.

On the one hand, common research in the field of simulated perception sensors has mainly been motivated by the implementation and validation of Advanced Driver Assistance Systems (ADAS) [6, 8, 12]. This results in detailed models trying to closely mimic the behavior of the physical sensors. Hence a vehicle simulator might be the best choice for realization. Yet, generating such sensor data with a vehicle simulator and handling them in application models is computationally expensive and might be only applied, when needed. On the other hand, there are many applications, which rely on pre-processed data like *Object Information* instead of *Raw Data* like sensor images. The object-related data may contain the positions, dimensions, and directions of other traffic entities. In turn, research questions, about how such applications improve traffic efficiency, might be investigated in large-scale scenarios. To enable this is the main goal of this paper.

Accordingly, this paper presents a Perception Model on Object-Level in the MOSAIC Application Simulator to be used in combination with large-scale traffic simulations. The implementation aims for a generic solution with a consistent interface for applications supporting the seamless exchange of different simulators such as Carla, PHABMACS, and SUMO, which are all coupled to MOSAIC. Yet, it uses most of the features of the traffic and vehicle simulators themselves. Specifically for this paper, the implementation details are presented on the basis of SUMO and in turn on SUMOs Traffic Control Interface (*TraCI*) and the newer and faster solution *Libsumo*, as well as the available Context Subscriptions to retrieve relevant object information. These objects will then be maintained by the Application Simulator in a spatial index for fast search of objects in sight of view. Currently, the solution supports the most relevant case that vehicles perceive vehicles, meaning moving participants perceive moving participants, which implies the highest requirements for the spatial search. We choose a simplified perception model for the start. Error models or occlusion of traffic objects are not in the scope of this paper but will be part of further work.

The paper is structured as follows. At first, Section 2 discusses data structures and methods for fast handling (update and search) of spatial objects. Section 3 presents how the new perception module is integrated into Eclipse MOSAIC. Specifically, it covers the implementation in the Application Simulator, but also the interfaces for coupling SUMO. In Section 4, the implementation is evaluated regarding performance. In fact, we implemented several solutions, which are compared in different scenarios. Finally, Section 5 concludes the paper and gives an outlook for ideas of further integration and also of using the perception module for investigations.

2 Spatial Search

In the context of mobility simulation, all objects have a specific location on a 2-dimensional plane (ignoring the altitude). Those objects are static or can move on the plane. Perceiving other objects (e.g. from the viewpoint of a vehicle) requires an algorithm to find all objects within a given area. On a 2-dimensional plane, the problem of finding localized objects within a range can be solved using spatial search algorithms. There mainly exist two different types of spatial search:

- Finding the k-nearest neighbors given an arbitrary point in the search space.
- Finding a set of objects or points within a given range, which could be a rectangle, a circle, or any other geometrical shape.

In order to model a perception module supporting moving objects, it requires us to find all vehicles (or any other object) within the sight of any other vehicle. Therefore, an algorithm for a spatial range search needs to be considered.

In a very naïve approach, this could easily be solved by checking all simulated objects in a loop. This approach, however, would become increasingly slow with large object quantities, as for every vehicle all other existing vehicles (and objects) need to be considered. This always results in a complexity of $O(n^2)$ for n objects. Much faster approaches employ some form of spatial index to speed-up queries. In general, spatial indexes work by grouping spatially close objects in some form. A common approach is to use spatial trees, such as K-D-trees, R-trees[5], or Quad-trees. These data structures allow a fast search of localized objects, resulting in a complexity of $O(\log(n))$. A major problem with these kinds of data structures is, however, that they are fast at finding points, but not at updating the structures when objects are moving. Furthermore, the issue of moving objects can be solved by using a grid-based data structure, which stores the objects in cells spanned across the map.

The choice of the best suited spatial index highly depends on the type of data, the amount of data, the dimensionality of the data, and the dynamics of the use case. Therefore, a lot of research has been conducted to compare different spatial indexes on different datasets. For example, Kothuri et al.[7] conducted an in-depth comparison of Quad-trees and R-trees on spatial data and concluded that index building and update operations are faster in Quad-trees, while query operations perform faster on R-trees on average. In search of a well-suited spatial index for our purposes, we will further describe and evaluate those approaches.

2.1 Range Search in Spatial Trees

Trees are a simple yet effective structure to find specific data points in a given set. For example, binary search trees are used to find one-dimensional data, such as objects in a hash table. For spatial data, e.g. localized objects within a 2-dimensional plane, a special kind of tree is required to gain similar results. All existing tree-like structures used for spatial search follow the *branch-and-bound* principle, which arranges data points within certain bounds on different levels. These bounds can be used during a search in the tree to discard branches that lay outside the range to search in.

An **R-tree** groups nearby points and creates bounding rectangles around those. This procedure is repeated within each bounding rectangle, representing a second level of the search tree. Depending on the number of points existing in the search space, this is done multiple times resulting in m levels. During search for a range (or single point), only those bounding rectangles of the first level which contain the search range are considered for deeper search.

A **K-D-tree** shows a very similar structure. Instead of bounding rectangles, a median line on one axis through the search space is calculated, dividing the set of points into two halves. This is again done for each of the two halves but using the other axis. The whole process is executed several times depending on the number of points, resulting in different levels and branches to be used during search. The main disadvantage compared to the R-tree is, that no items can be added or removed without re-building the entire tree.

In a **Quad-tree**, the search space is divided into four quadrants. This is again done recursively for each quadrant, depending on the number of points inside. This results in a tree with nodes having zero or four children each. Similar to the R-tree, insert and remove operations can be implemented efficiently. However, if the resulting tree is quite unbalanced (i.e. the points in the search space are unevenly distributed), the complexity of all operations search, insert, remove, can increase up to $O(n)$. Therefore, the Quad-tree requires a suitable configuration of the size of the quadrants (tile size), which is determined from the initial bounds the tree is covering.

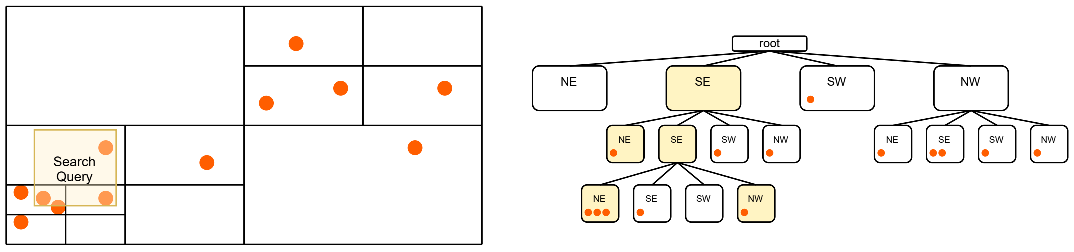


Figure 1: Example of a Quad-tree with a split size of 4. Only those quad-tiles which intersect with the search query are considered.

All tree variants described above are optimized on search, but lack on updating the objects in the search space. The K-D-tree requires a complete re-build when adding or removing objects, and the R-tree requires the bounding rectangles to be continuously updated when objects are moving. The Quad-tree, however, uses fixed tile sizes and could be a good candidate as it gives room for improvements that are suitable for our use cases. Therefore, we propose to use a Quad-tree like structure, with the additional properties (see Fig. 1):

- Each node of the Quad-tree stores multiple objects. Only if a maximum number of objects per node (*split size*) is exceeded, the node is split and all objects are distributed into four child quad tiles. Also, if all child quadrants of a node together contain less than a minimum number of objects per node (*join size*), all child nodes are joint. The actual values for *split size* and *join size* might depend on the expected total number of objects present in the search space.

- The previous improvement also supports moving objects. Therefore, we check if the moved object is still within the bounds of its tile. Only if it is outside, the object is completely removed from the tree and added again.

2.2 Range Search Using Spatial Grid

A different approach to index spatial data is using a grid-like structure instead of t rees. In this case, the search domain is split by a grid into equally sized cells. To address cells by given coordinates, the x and y-coordinates are converted accordingly. The formula for this conversion depends on the size of the grid and the area bounding the spatial data to search for. Adding or moving data points can be done in $O(1)$ time, as the cell address can be derived from its coordinates. To find data, all cells intersecting with the search range need to be considered, and all data points within those cells need to be checked (see Fig. 2). Therefore, the performance of this index depends on the grid size and the number of data points in each cell. One disadvantage of this data structure is memory consumption, as cells are always allocated even if they do not contain any data points. This could be problematic, especially in large-scale scenarios. When using range queries, the size of the search area (e.g. a rectangle or circle) affects the number of cells to be considered. It is advised to use range queries not larger than one cell, limiting queries to four grid cells at maximum.

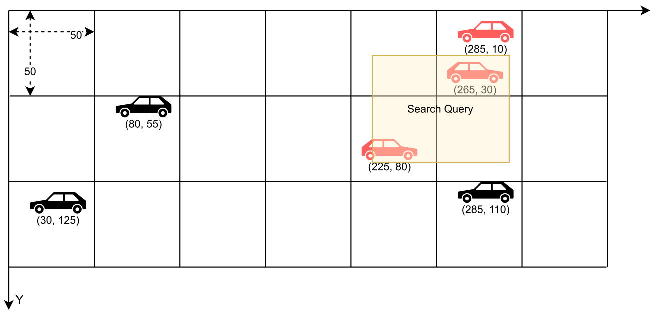


Figure 2: Searching spatial items using a grid index with a cell size of 50. Objects in cells are selected according to the search query.

2.3 Finding Traffic Objects in Sight of View

To model a perception facility, we define a field of view for every vehicle. All other traffic objects within this field of view should be detected by this model with the aid of the previously described methods for spatial search. Any occlusion or error models are neglected here but could be added as an additional filter. The field of view is defined by a sight distance h and a view angle γ, resulting in a circle around the center of the vehicle bound by two vectors (see Fig. 3). The perception is done in two steps:

1) An axis-aligned minimum bounding rectangle (MBR) around the field of view is calculated. Using the sight distance h as the radius of a circle around the center position of the vehicle, vectors b and c are calculated using the opening angle γ. Furthermore, any of the four additional vectors $\vec{a_0}, \vec{a_1}, \vec{a_2}$, and $\vec{a_3}$ with a length of h in both directions of both axes are considered, if they lie within the opening angle γ (see Fig. 3). Based on these vector points, the MBR can be easily calculated and is then used to query a range search in the given spatial index.

2) For all items returned by the spatial range search we check if they are inside the field of view of the vehicle. For each traffic object m at relative location $\vec{m}$, we use the dot product with vectors $\vec{b}$ and $\vec{c}$ to determine if object m lies inside or outside the bounds (Eq. 1 and 2). This approach has the advantage, that it does not rely on trigonometric functions during requests and can therefore be executed faster. However, this method only works for viewing angles $< 180°$. An additional check by comparing the magnitude of $\vec{m}$ with the sight distance h reveals if the object m is visible or not (Eq. 3).

$$\vec{c} \cdot \vec{m} \geq 0 \tag{1}$$

$$\vec{m} \cdot \vec{b} \geq 0 \tag{2}$$

$$|\vec{m}| \leq h \tag{3}$$

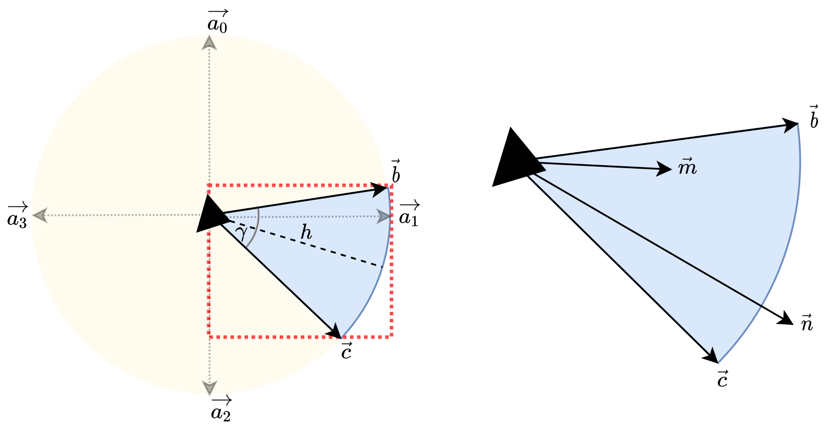


Figure 3: Field of view as a circle bound by two vectors $\vec{b}$ and $\vec{c}$ derived from sight distance h and a view angle γ. Left: Determining an axis-aligned minimum bounding rectangle (MBR, red) to use as search query. Right: Using vectors $\vec{b}$ and $\vec{c}$, and the sight distance h from the center of the vehicle, it can easily be checked if traffic objects at $\vec{m}$ or $\vec{n}$ are visible.

3 Implementation in Eclipse MOSAIC

The implementation of a perception module in the simulation framework Eclipse MOSAIC affects various components. To comprehend our design decisions, we give a brief overview of the concepts used in the MOSAIC framework. MOSAIC includes a middleware written in Java, which couples various simulators with each other. This middleware, the runtime infrastructure (RTI), uses *interactions* to exchange data among all coupled facilities. The coupling of an individual simulator is following the *federate* and *ambassador* principle, in which the actual simulator is wrapped by a federate interface that exchanges information with the ambassador. The ambassador is therefore coupled directly with the RTI by implementing and employing provided interfaces. A strict separation of the federate and ambassador is not mandatory, as a simulator that is already written in Java can be directly integrated into an ambassador implementation.

Modeling of smart mobility applications for different entities is done in the Application Simulator bundled with Eclipse MOSAIC. The movements of individual vehicles are usually modeled in a vehicle or traffic simulation, e. g. with PHABMACS or Eclipse SUMO. In order to provide a perception module for individual applications, the Application Simulator requires positions of all perceivable objects, especially vehicles in the traffic simulation. The following subsections briefly describe both the application simulator and the SUMO coupling, before continuing with the actual implementation of the perception module.

3.1 The MOSAIC Application Simulator

Eclipse MOSAIC supports applications for several entities such as vehicles, pedestrians, traffic signals, RSUs, servers, traffic management centers, and charging stations. The MOSAIC Application Simulator, therefore, provides an API to easily integrate custom application models. It allows, for one thing, using various facilities available on the respective unit (see Fig. 4). For example, applications have access to a communication module for building and sending V2X messages. Additionally, vehicles are provided with a navigation module to calculate routes on the road network. Furthermore, application code can read vehicle data, such as position, speed, heading, or sensor data, such as the distance to its leader. This information is usually derived from the traffic or vehicle simulator coupled with MOSAIC, e.g. SUMO. Additionally, many vehicle-actions can be triggered, e.g. stopping or re-routing the vehicle, or changing speed and lanes.

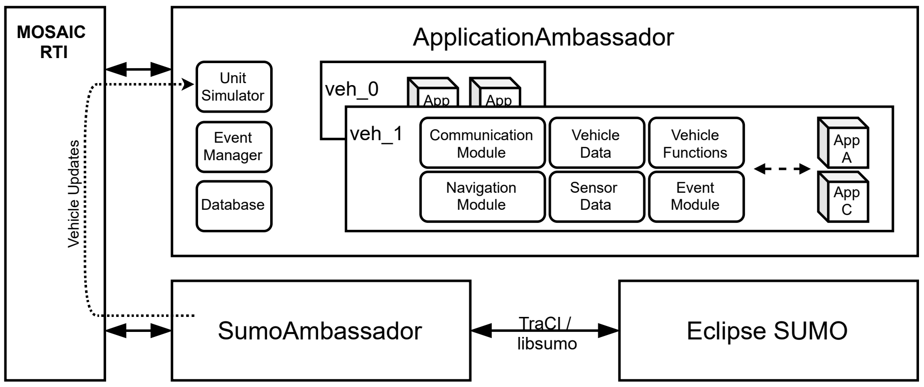


Figure 4: Overview of the Application Simulator bundled with Eclipse MOSAIC.

3.2 Coupling Traffic with Eclipse SUMO

In the context of MOSAIC, the traffic simulation of SUMO is used to handle the movements of individual vehicles. A coupling interface is used to realize the integration since MOSAIC is written in Java and SUMO in C++. In the current version, the *Traffic Control Interface* (TraCI) is used for this purpose. This socket-based protocol allows exchanging information between SUMO and any other application. The protocol is well-documented and could be easily adapted by the contributors of MOSAIC and is used for many years. However, the main issue of this coupling method is its performance, as it hardly depends on the I/O capabilities of the machine SUMO is running on. To reduce I/O overhead, SUMO uses the concept of *subscriptions*, which allows the client to define in advance which data from which vehicles should be sent via *TraCI* after each simulation step. This indeed reduces much of the protocol overhead, but the requested data still must be sent via the socket.

The *SumoAmbassador*, which implements the *TraCI* client on the MOSAIC side, could already be configured in a way, that only vehicles equipped with MOSAIC applications are subscribed. This is helpful in many use cases, especially in those with low penetration rates. However, when thinking of perception, the position of all entities (i.e. vehicles) is again required to insert them into the search space, which would require basic position data for all vehicles to be subscribed.

As an alternative to *TraCI*, recent developments introduced *Libsumo* as a second method to integrate SUMO into Java- or Python-based programs. This library makes use of the SWIG framework to create bindings for different programming languages, such as Java, allowing to integrate SUMO as a native library using the Java Native Interface (JNI). With this method being used, data is not sent via a socket anymore but is read directly from the memory. This integration method has been adopted by the MOSAIC coupling implementation of SUMO, enabling faster simulations, especially when subscribing to many vehicles.

The requirements to use the perception module by MOSAIC applications include, that the positions of all present vehicles must be known at any time. This implies that the coupling implementation of SUMO with MOSAIC has to request basic data for all vehicles, such as position, heading, and speed. To achieve this, basic subscriptions for all vehicles are used in the *TraCI* implementation or read directly from SUMO using the *Libsumo* implementation. In that context, SUMO already comes with a feature that could solve the problem of object perception already without the need for dedicated spatial indexing and search in the MOSAIC Application Simulator. With *context subscriptions*, any client can subscribe for data of all surrounding vehicles for a given ego-vehicle. Additionally, SUMO can already filter surrounding vehicles by defining a field-of-vision, which is expected to give equal results as our own perception model. This indeed would reduce the amount of data to read, as only (single and context) subscriptions for equipped vehicles would be required. However, the main goal is to provide a unified API in the application simulator regardless of the traffic or vehicle simulator used. Therefore, we decided to implement two solutions: 1) makes use of context subscriptions and the field-of-vision filter in SUMO, and 2) implements our own perception module which is fed with basic position data of all vehicles in the simulation. In Section 4.2 the solution of using context subscription is directly compared with our perception module, which is described in the following.

3.3 Perception Module

The perception module is integrated into the MOSAIC Application Simulator in a way, that each application has access to it. As a result, every entity can request surrounding objects in its field of view. Next to vehicles that perceive other traffic participants, this allows modeling road side units or traffic signals having access to a sensor detecting other traffic objects. The API usage from inside an application can be read in Listing 1.

The spatial index itself, be it Grid or Quad-tree, is only instantiated once and can be accessed globally by all active perception modules. To avoid unnecessary update calls, e.g. if the perception is called rarely or not used at all, the index is only updated on request (lazy loading pattern). To achieve this, we provide a parameter `spatialIndexUpdateInterval` to define how long (in simulation time) the index is used for requests without being updated. Each spatial search request comes with a `PerceptionRange` parameter, which provides a bounding rectangle as the search range query. The result of the index is then filtered by the perception module according to its field of view (see Section 2.3).

The actual implementation of the spatial index variants Grid and Quad-tree can be found in the Eclipse MOSAIC repository [1]. The implementations are interchangeable (see Fig. 5) as the best-performing index might be chosen dependent on scenario size or view range. A hash table is used for both implementations to fast update the actual traffic objects in the index. The index itself is updated once the location of all traffic objects has been changed. If the grid cell index of the moved object has not changed, or it is still within the bounds of its current quad-tile respectively, then no update operation for that particular object is performed.

Listing 1: Activating and using the perception module in a MOSAIC application.

```java
public class PerceptionApp extends AbstractApplication {

  @Override
  public void onStartup() {
    getOs().getPerceptionModule().enable(
        new SimplePerceptionConfiguration(60.0, 200.0)
    );
  }

  @Override
  public void onVehicleUpdated(VehicleData previous, VehicleData updated) {
    List<VehicleObject> vehicles =
        getOs().getPerceptionModule().getPerceivedVehicles();
    // do something with the perceived vehicles
  }

}
```

To optimally perform in different scenarios the Quad-tree can be parametrized. A quad-tile is not split until more objects than `splitSize` are stored in that particular tile. If `splitSize` is exceeded, four more quad-tile are created and all objects are spread over these according to their positions. To avoid unnecessary split and join operations, the parameter `joinSize` defines when four quad-tiles are joined together. This parameter must be lower than `splitSize` to function. Furthermore, the `maxDepth` parameter is used to determine how many levels of split quad-tiles can exist at maximum. If the maximum depth is reached, a quad-tile can hold more than `splitSize` objects.

Parameters for the Grid index only adjust the width and height of each cell. Therefore, the actual size of the grid (number of cells) varies with the scenario boundary. The four corners of the bounding rectangle of the search query are used to find the minimum and maximum row and column indexes. All objects of the cells within these bounds are collected and filtered to fit into the search range query.

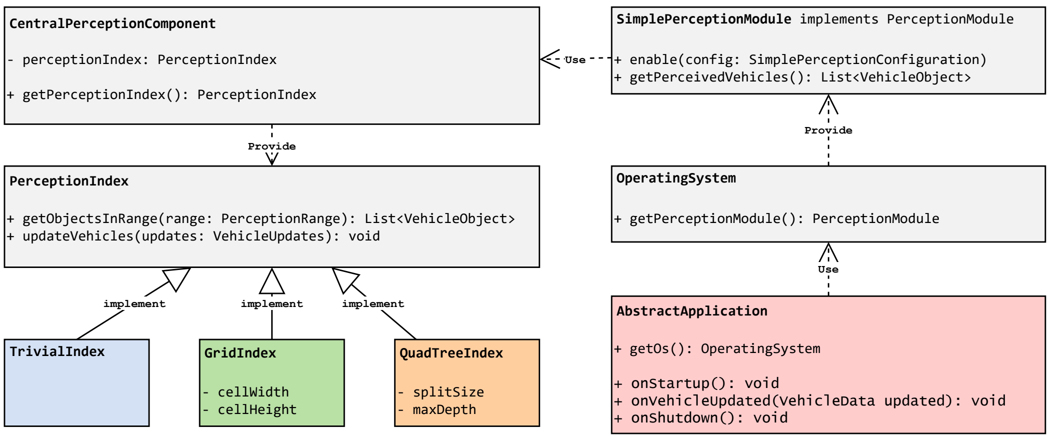

CentralPerceptionComponent
- perceptionIndex: PerceptionIndex
+ getPerceptionIndex(): PerceptionIndex
Use
SimplePerceptionModule implements PerceptionModule
+ enable(config: SimplePerceptionConfiguration)
+ getPerceivedVehicles(): List<VehicleObject>
Provide
Provide
PerceptionIndex
+ getObjectsInRange(range: PerceptionRange): List<VehicleObject>
+ updateVehicles(updates: VehicleUpdates): void
OperatingSystem
+ getPerceptionModule(): PerceptionModule
Use
implement
implement
implement
TrivialIndex
GridIndex
- cellWidth
- cellHeight
QuadTreeIndex
- splitSize
- maxDepth
AbstractApplication
+ getOs(): OperatingSystem
+ onStartup(): void
+ onVehicleUpdated(VehicleData updated): void
+ onShutdown(): void

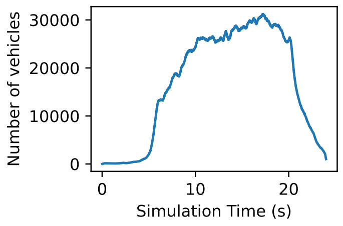
Figure 5: Class structure of the perception integration in the MOSAIC Application simulator.

4 Evaluation

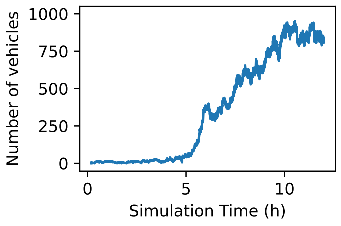
In this section, we compare the performance of the spatial indexes implemented in MOSAIC. In addition, we also compare our solution with built-in features in SUMO to find out which implementation is more suitable for specific use cases. For our experiments, we employ various simulations with MOSAIC coupling the traffic simulation of SUMO. For the actual simulation of the traffic, we defined two simulation scenarios of different sizes. Firstly, we use a full-day city scenario with several million vehicles in total, in order to measure the performance on a large scale. Secondly, we use an inner-city scenario with several hundreds of vehicles with varying traffic demand, which is a more common size for a mobility simulation.

The **Berlin** scenario contains traffic demand with around 1,8 million trips during 24 hours within the whole city of Berlin, Germany. The traffic demand was generated by extracting 80 % of vehicle trips from the MATSim Open Berlin [13]. Routes for these trips have been iteratively calibrated using the `duaIterate.py` script[1]. This scenario is furthermore planned to be published under an open-source license on GitHub [3].

The **Charlottenburg** scenario covers a smaller area of Berlin. It was created by cutting out traffic from the Berlin scenario using the `cutRoutes.py` script[1]. The simulation duration was reduced to 12 hours including the morning peak only. In total, this scenario contains 67 000 vehicle trips, with 900 vehicles being present simultaneously at maximum (see Fig. 6).



(a) Berlin



(b) Charlottenburg

Figure 6: Traffic volumes modelled for each simulation scenarios.

Each simulation scenario was configured to run with MOSAIC in order to examine the different spatial index implementations. The perception module in MOSAIC can only be used by applications mapped to individual vehicles. We therefore configured a mapping to deploy a simple application that solely requests the surrounding vehicles within its field of vision in every simulation step (= every second) (see Listing 2). The field of vision was configured with a 200 m range and an opening angle of 60°.

Listing 2: Mapping configuration to deploy a perception module on 10 % of the vehicles.

```
{
  "prototypes": [ {
      "name": "DefaultVehicle",
      "applications":[
        "org.eclipse.mosaic.app.tutorial.vehicle.PerceptionApp"
      ],
      "weight": 0.10
  } ]
}
```

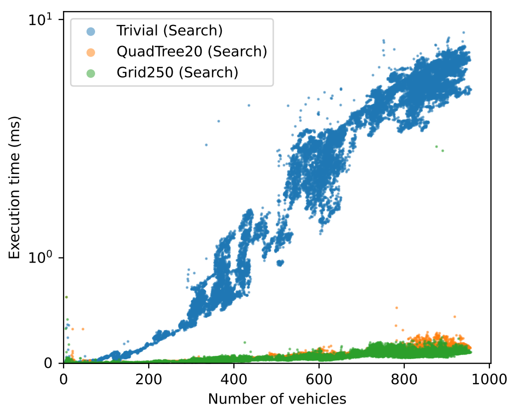
[1]This script is delivered with the SUMO installation [9].

4.1 Performance of Spatial Indexes

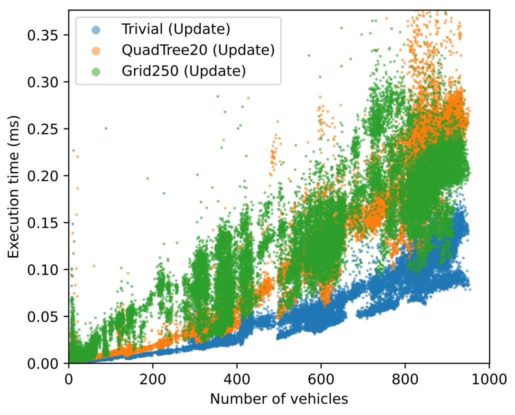
For the performance evaluation of the different index implementations, we defined the following configurations to compare against each other:

- **Trivial** - No spatial index. A view check is done for all existing vehicles in a for-loop.
- **QuadTreeXX** - Quad-tree implementation with a configuration of a maximum depth of 12, and a maximum number of vehicles per leaf/tile of [10, 20, 30, or 40].
- **GridYYY** - Grid-based index implementation having different cell sizes. The cell size is either [50, 100, 250, or 500] m in length and height.

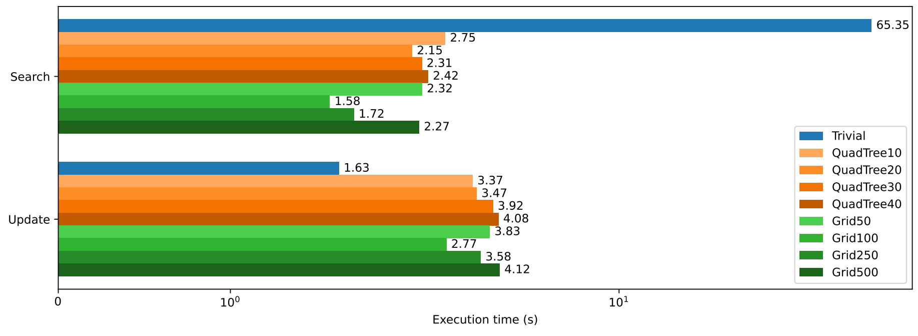
For these experiments, 10% of the vehicles are equipped with the perception module. Each equipped vehicle requests the vehicles in its field of vision in every simulation step. During simulation, we measure the duration of every call of the atomic operations *update*, *search*, and *remove*. We furthermore store the current number of vehicles present in the simulation for each measurement, in order to find a relation between duration and index size.



(a) Search operations (logarithmic scale)



(b) Update operations (linear scale)

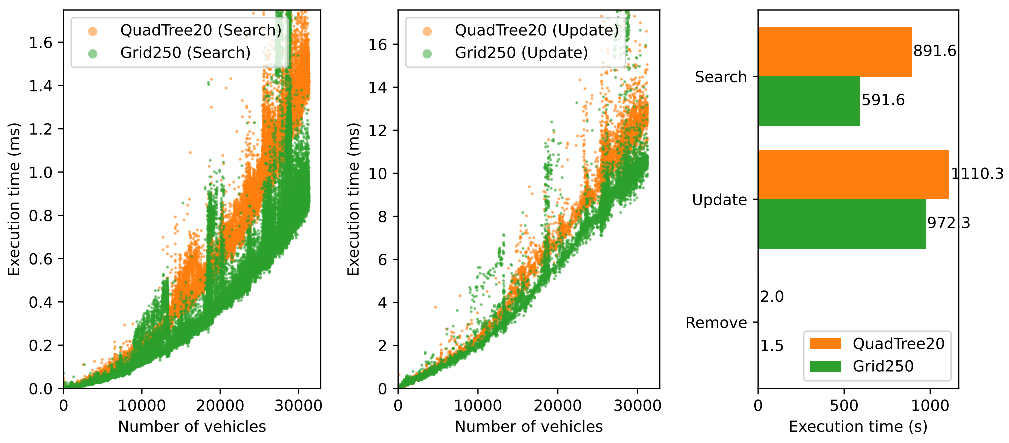


(c) Accumulated durations (logarithmic scale)

Figure 7: Individual and accumulated durations of search or update operations on each spatial index implementation in the Charlottenburg scenario.

For a first comparison, we run the Charlottenburg scenario with the nine different index configurations. Results can be found in Figure 7. Not surprisingly, the search queries require a lot of time in the trivial approach compared to Grid or Quad-tree. Even in situations with a low number of vehicles (e.g. less than 200), a spatial index is already much faster. In general, Grid and Quad-tree show very similar performance results. Search and update performance of the Grid is slightly better than the Quad-tree. Having a look at the individual configurations show, that both, lower tile capacity in the Quad-tree and very small cell sizes in the Grid, have disadvantageous effects on the performance. Best results can be achieved by using a Quad-tree with a maximum capacity of 20 or 30 vehicles per tile, or a Grid with a medium-large cell size of 100 or 250 m. Choosing a cell size also most likely depends on the viewing range, as it affects the number of cells in the Grid to be initially chosen for the range check. However, we did not investigate the effects of the viewing range on the performance of the Grid. In comparison to the total simulation time of 240 seconds, the overhead of 4–7 seconds produced by the spatial index is minimal and does not require further examination.

In a second series of experiments, we configured the Berlin scenario with the Quad-tree and Grid implementation, and again an equipment rate of 10%. Since this massive scenario already requires several hours to complete, we skipped the Trivial approach. Figure 8 shows that the Grid can outperform the Quad-tree in large-scale scenarios. Search operations with the Grid are 30% faster, update operations show 10% better performance. The total overhead of the index for search and update operations compared to a total simulation time of roughly nine hours is found to be at 5% for the Grid, and 6% for the Quad-tree.



(a) Search operations (b) Update operations (c) Accumulated durations

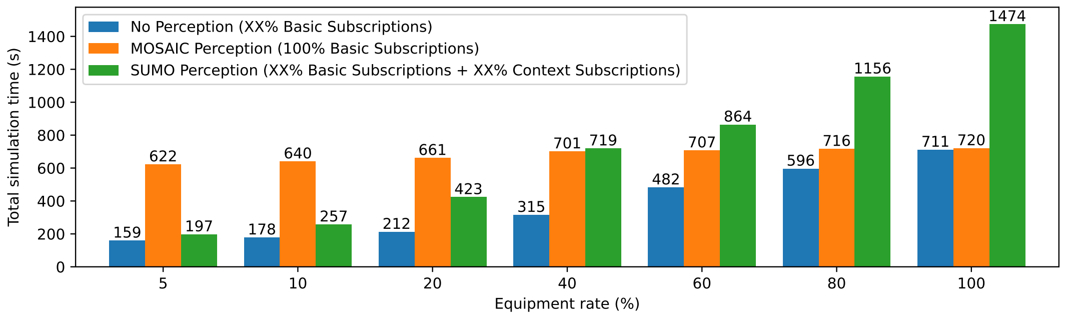
Figure 8: Durations of update and search operations in the Berlin scenario.

4.2 Performance of Coupling with SUMO

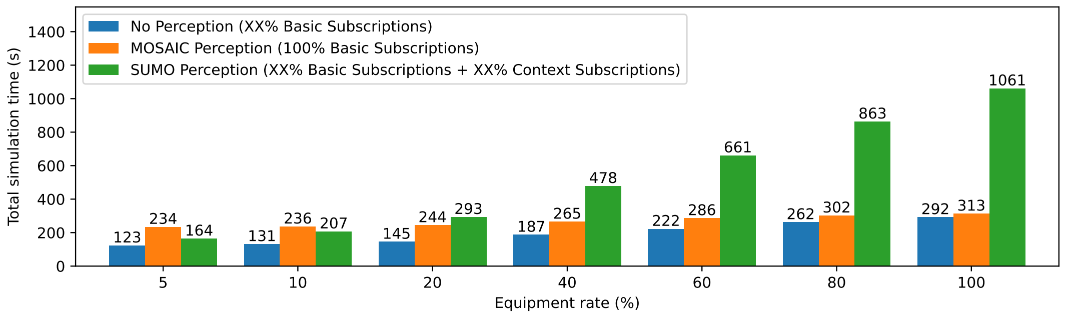
The following experiments help to understand the overhead that comes with the coupling of the traffic si mulation of SU MO. Th is co mparison in cludes tw o as pects. Fi rstly, th e tw o different coupling interfaces using *TraCI* or *Libsumo* are compared with each other. Secondly, our implementation of the perception module is compared with the built-in features provided by the interfaces of SUMO (i.e. context subscriptions), as explained in Section 3.2. We, therefore, define the following simulation runs, which will furthermore be executed using the two different coupling methods:

- **No Perception -** As a baseline, basic vehicle data is subscribed for [5, 10, 20, 40, 60, 80, or 100]% of the vehicles, which are equipped with any application. No perception index is used at all. This allows measuring the total overhead each of the subsequent solutions requires.
- **MOSAIC Perception -** Basic vehicle data is subscribed for *all* vehicles in the simulation. [5, 10, 20, 40, 60, 80, or 100]% of the vehicles are equipped with a perception module using a QuadTree20 as a reference for a spatial index implementation.
- **SUMO Context Subscriptions -** Basic vehicle data is subscribed *only* for equipped vehicles. Additionally, context subscriptions are used for equipped vehicles to retrieve vehicles within the field of vision. The perception module in MOSAIC uses the results of these context subscriptions rather than building its own spatial index. The equipment rate is either [5, 10, 20, 40, 60, 80, or 100]%.

The following experiments include simulations of the Charlottenburg scenario. The equipment rate is varied from 5, 10, 20, 40, 60, 80, to 100%. As a coupling method, *TraCI* is compared to *Libsumo* to find the performance improvement this new coupling method comes



(a) TraCI



(b) Libsumo

Figure 9: Total simulation duration using *TraCI* and *Libsumo* coupling under different equipment rates in the Charlottenburg scenario.

with. Figure 9 shows, that with an increasing equipment rate, the simulation time of all variants also increases. The solution which uses the spatial data structures in MOSAIC (i.e. MOSAIC Perception) exhibits a certain time overhead at all equipment rates. The main reason here is the required subscriptions of all vehicles in the simulation, which need to be transferred to MOSAIC, in case (a) *TraCI* via TCP. Using SUMOs context subscriptions to obtain surrounding vehicles (i.e. SUMO Perception) shows a large benefit at lower equipment rates, but inferior scalability compared to MOSAIC Perception at higher equipment rates. Eventually, in the 100% case the SUMO Perception variant requires twice the time as running SUMO without any context subscriptions, and almost ten times as long as the 5% case.

Using the new *Libsumo* coupling interface on the other hand can improve simulation time significantly (see case (b) in Fig. 9). The simulation time of the MOSAIC Perception case can be reduced almost by a factor of three. Again, the difference between a low and high equipment rate is minimal in the MOSAIC Perception case and only depicts the overhead produced due to application handling and utilizing the spatial index. Comparing with the SUMO Perception variant, it is already enough to deploy 20% of the vehicles to find better performance with the MOSAIC Perception. Still, the SUMO Perception case increases simulation time with the equipment rates.

4.3 Evaluation Summary

In various conducted experiments, we investigated two aspects of the perception module implementation in MOSAIC. For one thing, a detailed examination of the index implementations Grid and Quad-tree and their configuration options showed, that both variants work well without producing much computational overhead. The Grid performs slightly better, especially in large scenarios with several thousands of vehicles being simulated in parallel. Here we found, that middle-sized cells of 100 to 250 m work best. The ideal cell size, however, is expected to depend on the view range and angle as this affects the number of cells being selected during search. Nevertheless, the Quad-tree implementation still shows good performance even with a large amount of moving objects in the index, for both update and search operations.

Another aspect is the interaction with SUMO. The amount of data captured from SUMO during simulation has an immense influence on the overall performance of the simulation, especially since the perception module implementation in MOSAIC requires the positions of all existing vehicles. Here we found, that using *TraCI* results in large overhead, which can mostly be eliminated by using the new *Libsumo* interface.

However, we also found that the built-in feature of SUMO, which is able to obtain surrounding vehicles as well, shows decreased performance on higher equipment rates. A deeper look into the actual MOSAIC simulation using a profiler revealed, that most of the additional time comes not only with the additional results from context subscriptions to read. The simulation call itself, using `Simulation.simulateUntil()` requires more than twice as much time as soon as context subscriptions are added for 40% of the vehicles. Additional time is then spent read-

Experiment	**Total**	**Simulate Step**	**Read single subscriptions**	**Read context subscriptions**	**MOSAIC overhead**
40% Single subscriptions	187 s	107 s	58 s	0 s	22 s
40% Single subscriptions + 40% Context Subscriptions	478 s (+ 156%)	259 s (+ 142%)	60 s (+ 3%)	133 s (+∞%)	26 s (+ 18%)

Table 1: Increase of simulation duration due to use of context subscriptions in *Libsumo*.

ing the context subscription results using `Vehicle.getAllContextSubscriptionResults()`. Detailed results of this brief analysis can be found in Table 1.

5 Conclusion and Outlook

Extracting object information from raw sensor data (e.g. Camera) and feeding them into smart mobility applications can help to improve safety and traffic efficiency. Co-simulation frameworks such as Eclipse MOSAIC may be used to develop and test such applications. In order to enable object detection on an application level, the integrated models need to provide information about surrounding objects. Such objects can be stationary (e.g. traffic signs, traffic signals), or moving (e.g. vehicles, pedestrians). Obtaining moving objects within a field of view of a stationary or other moving traffic participant requires a fast method to prevent low-performing simulations.

In this paper, we solved this problem by introducing a perception module to the Application Simulator of MOSAIC. For finding moving objects within a given range (e.g. inside the field of view of another vehicle), we implemented and evaluated different spatial index implementations based on a grid or Quad-tree. We evaluated the implementation with large-scale simulation scenarios in combination with the traffic simulation of Eclipse SUMO, with a view to the computational performance. The perception method using a grid to find surrounding objects worked best, with varying performance depending on the size of the included cells. Using a Quad-tree instead resulted in slightly but not significantly slower simulations. The modeling of perception in MOSAIC required reading basic data (e.g. position) for all moving objects (here vehicles) from SUMO during the simulation. This led to a bottleneck when the socket interface *TraCI* was still used to exchange data. In the course of this paper, we improved the integration of SUMO by implementing the *Libsumo* interface. Evaluations showed, that with *Libsumo* the simulation time could be improved by a factor of three in the presented scenarios. Only with the use of *Libsumo* feasible simulation times are possible in combination with our perception module implementation.

In following work, we want to use object perception for improving traffic efficiency. Here we believe that object detection through cameras in vehicles could be used to pre-process and pre-estimate traffic states on road segments. Such pre-estimated data could be shared and used to improve overall traffic state estimation, which is currently only achieved by collecting floating-car-data from a large fleet of vehicles.

In the current implementation of the perception module only moving objects have been considered. This work will be published with the next release of Eclipse MOSAIC and is already integrated into the source code repository in [1]. Future work will also allow the perception of stationary objects, such as traffic signs and signals. Furthermore, additional models which filter the set of surrounding objects due to occlusion are subject to ongoing work. The traffic scenario used in the evaluation section of this paper is planned to be published under an open-source license on GitHub soon [3].

Acknowledgment

This work is part of the “eBaseCamp” project (funding number 01ME19002C), which is funded by the Federal Ministry for Economic Affairs and Climate Action (BMWK), Germany. We furthermore want to thank Fabian M. Thiele and Trung Han Nguyen for their implementation work on the spatial index implementations.

References

[1] MOSAIC contributors. Eclipse MOSAIC Source Code. Eclipse Foundation on Github, [online], 2020-2022. https://github.com/eclipse/mosaic.

[2] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. In *Conference on robot learning*, pages 1–16. PMLR, 2017.

[3] Eclipse MOSAIC Core Team. Berlin SUMO Traffic (BeST) Scenario. https://github.com/mosaic-addons/best-scenario, 2022.

[4] Eclipse MOSAIC Core Team. Eclipse MOSAIC: A Multi-Domain and Multi-Scale Simulation Framework for Connected and Automated Mobility. https://eclipse.org/mosaic, 2022.

[5] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[6] Maike Hartstern, Viktor Rack, Mohsen Kaboli, and Wilhelm Stork. Simulation-based evaluation of automotive sensor setups for environmental perception in early development stages. In *2020 IEEE Intelligent Vehicles Symposium (IV)*, pages 858–864. IEEE, 2020.

[7] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 546–557, 2002.

[8] Clemens Linnhoff, Philipp Rosenberger, Simon Schmidt, Lukas Elster, Rainer Stark, and Hermann Winner. Towards serious perception sensor simulation for safety validation of automated driving-a collaborative method to specify sensor models. In *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, pages 2688–2695. IEEE, 2021.

[9] Pablo Alvarez Lopez, Michael Behrisch, Laura Bieker-Walz, Jakob Erdmann, Yun-Pang Flötteröd, Robert Hilbrich, Leonhard Lücken, Johannes Rummel, Peter Wagner, and Evamarie Wießner. Microscopic traffic simulation using sumo. In *The 21st IEEE International Conference on Intelligent Transportation Systems*. IEEE, 2018.

[10] Kay Massow, Fabian Maximilian Thiele, K Schrab, BS Bunk, I Tschinibaew, and Ilja Radusch. Scenario definition for prototyping cooperative advanced driver assistance systems. In *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8. IEEE, 2020.

[11] Robert Protzmann, Björn Schünemann, and Ilja Radusch. Simulation of convergent networks for intelligent transport systems with vsimrti. *Networking Simulation for Intelligent Transportation Systems: High Mobile Wireless Nodes*, pages 1–28, 2017.

[12] Francisca Rosique, Pedro J Navarro, Carlos Fernández, and Antonio Padilla. A systematic review of perception system and simulators for autonomous vehicles research. *Sensors*, 19(3):648, 2019.

[13] Dominik Ziemke, Ihab Kaddoura, and Kai Nagel. The matsim open berlin scenario: A multimodal agent-based transport simulation scenario based on synthetic demand modeling and open data. *Procedia Computer Science*, 151:870–877, 2019. The 10th International Conference on Ambient Systems, Networks and Technologies (ANT 2019) / The 2nd International Conference on Emerging Data and Industry 4.0 (EDI40 2019) / Affiliated Workshops.