Component-Based Design of Cyber-Physical Applications with Safety-Critical Requirements

Alejandro Masrur[a], Michał Kit[b], Vladimír Matěna[b], Tomáš Bureš[b], Wolfram Hardt[a]

[a] *Department of Computer Science, TU Chemnitz, Germany*
[b] *Faculty of Mathematics and Physics, Charles University, Czech Republic*

Abstract

Cyber-physical systems typically involve large numbers of mobile autonomous devices that closely interact with each other and their environment. Standard design and development techniques often fail to effectively manage the complexity and dynamics of such systems. As a result, there is a strong need for new programming models and abstractions. Towards this, component-based design methods are a promising solution. However, existing such approaches either do not accurately model transitory interactions between components – which are typical of cyber-physical systems – or do not provide guarantees for real-time behavior which is essential in safety-critical applications. To overcome this problem, we present a component-based design technique based on DEECo (Dependable Emergent Ensembles of Components). The DEECo framework allows modeling large-scale dynamic systems by a set of interacting components and, in contrast to approaches from the literature, it provides mechanisms to describe transitory interactions between them. To allow reasoning about timing behavior at the component-description level, we characterize DEECo's closed-loop delay in the worst case, i.e., the maximum time needed to react to a change in the environment. Based on this, we incorporate real-time analysis into DEECo's design flow. This further allows us to analyze the system's robustness under unreliable communication and to design decentralized safety-preserving mechanisms. To illustrate the simplicity and usefulness of our approach, we present a case study consisting of an intelligent crossroad system.

Keywords: Cyber-physical systems, component-based design, safety-critical applications, real-time and timing analysis, unreliable communication, reliability-aware design

1. Introduction

Cyber-physical systems (CPS) are characterized by close interactions with their environment and can be found in different upcoming domains such as smart traffic and transportation, intelligent buildings, smart grid, etc. CPS are inherently distributed, i.e., they rely on a large number of autonomous, typically mobile, embedded devices that form an ecosystem. The joint operation of devices within this ecosystem provides functionality, which is otherwise unattainable by individual devices in isolation.

On the other hand, CPS are highly adaptive, i.e., they constantly react to changes in their environment by modifying their structure and/or behavior accordingly. The collaborative aspect between parts or components of such systems, as well as the necessity for them to function

autonomously (in case that, for example, the connection to other parts or components gets interrupted), poses an entire new dimension of challenges for designers. Typically, these challenges are regarded as separate problems of communication networks, distributed control, etc. As such, they have been addressed separately in the respective research fields. However, a significant aspect of modern CPS is that they often are *software intensive* [1], which still remains widely overlooked in the literature.

This means that most of their functionality is embodied in the software, which in turn becomes the most complex and critical constituent. Due to the fact of being distributed and adaptive, software becomes even more complicated and the system starts exhibiting so-called *emergent behavior*. This is the situation where the system's behavior cannot be inferred any longer from its individual parts or components, but their interplay and their joint influence on the environment have to be taken into account.

As a result, there is a strong need for *holistic* design and development methods that rather focus on the whole ecosystem and its overall behavior than on individual constituents. Especially, these methods have to provide systematic software engineering practices that allow managing the increasing complexity of such systems and help controlling emergent behavior. By systematic software engineering, we envision the following four aspects:

i) high-level (requirements-oriented) design with special focus on autonomous behavior, adaptivity and distributed collaboration,

ii) architectural design where a system is modeled by distributed components with clear responsibilities and well-defined interaction patterns,

iii) framework for implementation of components that allows for a straightforward traceability w.r.t. (i) and (ii), and

iv) methods for design-time and runtime analysis (e.g., functional verification, timing analysis) that predict and control the adaptivity and related emergent behavior of these systems.

The basic prerequisite for such systematic software engineering is the existence of software models providing an appropriate level of abstraction. In this respect, classical existing software models (e.g., component-based models such as AUTOSAR [2] or formal process models such as Petri Nets [3]) largely fail to address the needs of distributed adaptive systems. This is mostly because they rely on a static structure and thus are unable to model an *open-ended* system, which adapts its architecture to the current state of its environment.

On the other hand, new software models (such as DEECo [4], Helena [5]) appear gradually. They have been specifically developed to capture the nature of distributed adaptive systems and are more suitable for the design and development of CPS. However, by focusing on the cooperative aspects and dynamics of components, they *operate* at a high level of abstraction and do not provide sufficient means to model real-time behavior – which is particularly relevant to safety-critical applications.

Contributions. In this paper, continuing and extending our previous work [6], we bridge the gap between a (high-level) component-based description of the system and the analysis of its real-time behavior. In particular, we make use of DEECo (Dependable Emergent Ensembles of Components) in the context of a safety-critical cyber-physical application, viz., an intelligent crossroad system (ICS).

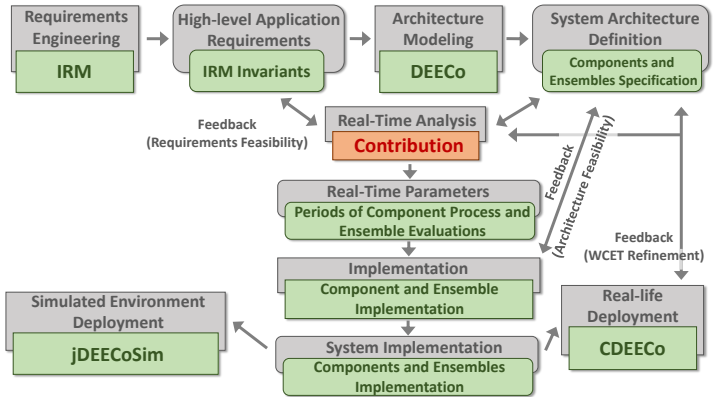

Figure 1: DEECo's design flow. A sharp rectangle represents a single step of the process. A rounded rectangle depicts outcome of a previous step. While, gray shapes correspond to the general design flow, the green ones are their concrete reifications into the DEECo framework.

DEECo allows for a component-based design of highly dynamic systems and provides deterministic semantics supporting real-time behavior. DEECo is a comprehensive solution as shown in Figure 1, which covers i) requirements engineering with IRM (Invariant Refinement Method) [7], ii) design and development based on DEECo constructs (i.e., components, ensembles, etc.) [4], iii) implementation and deployment with CDEECo (a C++ runtime environment) [8], and iv) simulation and test with the jDEECoSim simulator [9].

We characterize DEECo's closed-loop delay in the worst case, i.e., the maximum delay that it takes to react to a change in the environment. This builds the foundation for our real-time analysis of DEECo-based systems, which we illustrate in the context of our ICS. The results of this analysis are fed back to the component-description level which then capture the application's timing requirements. To this end, we extend DEECo's design flow as shown in Figure 1, where the proposed real-time analysis connects the high-level description with a concrete implementation of the system.

In contrast to [6], in this paper, we provide a more detailed treatment of DEECo's worst-case closed-loop delay and of the ICS case study. In addition, we analyze the system's robustness under unreliable communication and determine an upper bound on the number of packets that can be lost at the communication channel without compromising safety in our case study. Further, we discuss how to implement decentralized safety-preserving mechanisms, which are triggered when something goes wrong, e.g., communication is completely lost, etc. Finally, we validate our analysis by an extensive OMNeT++ simulation considering a varying number of packet losses at the communication channel.

The paper is structured as follows. Section 2 presents our case-study used as the running example, whereas Section 3 discusses related work. In Section 4, we give an overview of DEECo's basic concepts and illustrate them on our case study. Section 5 deals with DEECo's closed-loop delay in the worst case. This is then used to set up our real-time analysis as proposed in Section 6. In Section 7, we study the case of packet losses at the communication channel and discuss how to implement decentralized safety-preserving mechanisms for open-ended CPS. Following, in Section 8, we evaluate our approach by comparing it with results obtained by an OMNet++ simulation and, in Section 9, we present some conclusions.

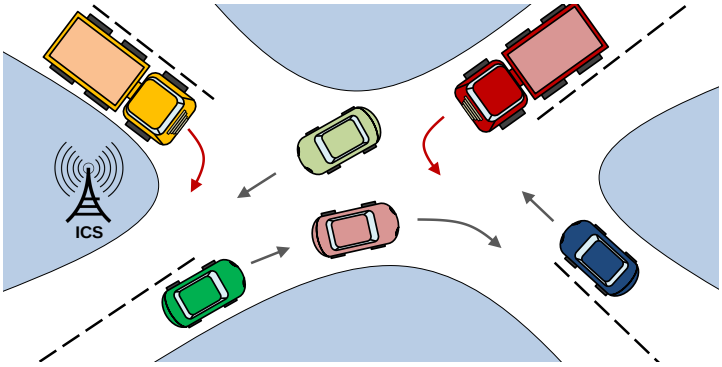

Figure 2: Intelligent crossroad system (ICS)

2. Case-Study

We consider an application scenario in the context of Vehicular Ad-hoc Networks (VANets) [10] and autonomous vehicles, where an ICS optimizes the *car throughput* at a road crossing. This is illustrated in Figure 2, where cars approach a two-lane crossing managed by the ICS.

The idea is to replace traffic lights to a great extent by using car-to-infrastructure (C2I) communication and synchronizing in what order cars cross the intersection for an uninterrupted flow in all directions. Note that there are different ways of implementing this case study. In particular, one can design the ICS to take full control over cars adjusting their speed and steering as considered in [6]. However, this concentrates almost all computation workload at the ICS – making more expensive hardware necessary – and requires cars to be enabled for remote operation.

In this paper, we follow an alternative approach, where the ICS computes – taking traffic conditions and regulations into account – and *assigns* a speed to an approaching car at the intersection. The car will then have to keep this speed constant to cross the intersection without stopping. This solution is more viable to implement the proposed case study, since it does not require cars to be modified for a remote operation. On the contrary, each (autonomous) car is responsible for driving along its own trajectory and maintaining its speed as assigned to it. If necessary, however, a car may stop to avoid a crash or accident.

The ICS assigns speeds to cars and these report their current speeds back to the ICS via C2I communication. As discussed later, the ICS needs to keep track of cars' speeds in order to detect potential hazards and trigger safety mechanisms. To this end, we foresee two operation modes:

i) Automatic mode: This is the default mode where all cars at the crossing behave as expected, i.e., they maintain their corresponding speeds as computed by the ICS. Here, the highest possible *throughput* is reached provided that speed limit regulations are observed and safety of all traffic participants can be guaranteed.

ii) Manual mode: This is the exception mode where one or more cars do not maintain their computed speeds and/or there is no communication between a car and the ICS, e.g., conventional cars with a human driver. In this mode, the ICS works as standard traffic lights.

We define the *region of influence* by the area in which the ICS *controls/monitors* all approaching cars. We assume that this area consists of a 50 m radius around the intersection. In this region,

vehicles are prioritized such that their priorities increase as they get closer to the center of the intersection and drop when they move away of it. Some of the vehicles at the ICS might also be *privileged* such as, for example, ambulances or police cars in an emergency situation, etc.

The ICS can detect when a car enters the region of influence, e.g., by radar, pressure sensors, etc. If no communication is received from one car after it has entered the region of influence (in particular, a car's intended direction, its current speed, etc. are needed), the ICS assumes that either there has been an error or it is a conventional car with no I2C communication and switches to manual mode. The same happens if communication is lost to one or more cars; a more detailed analysis of this is given in Section 7.

Pedestrians can be easily handled in the manual mode. In the automatic mode, the ICS can detect when pedestrians stand at the crossing for some time, e.g., by pressure sensors, request buttons, etc., and stop the traffic to let them cross in a safe manner. Again, each car is responsible for itself and should be able to react to unpredicted situations, e.g., performing an emergency break, according to valid traffic regulations.

This scenario exhibits different challenges that need to be faced when designing dynamic distributed systems. One of those challenges is the description of architectural changes that occur during runtime. In our case study, cars/vehicles arrive to and leave the system at different points in time, their priorities vary according to their distances to the crossing, etc. Such details need to be properly reflected in the system design.

Furthermore, this scenario exhibits real-time requirements imposed to the system. In particular, it is required that the *reaction time* between a car and the ICS is kept below a certain upper bound in order to ensure an appropriate responsiveness of the overall system, where unreliable communication needs to be considered. In turn, meeting those real-time requirements allows us to guarantee safety, which translates into a collision-free crossing in our case study.

Lastly, since it is not possible to guarantee a fully reliable communication, the system has to be designed to be self-adaptive. This way, the system switches to manual mode when it realizes that real-time requirements cannot be met. To this end, as discussed later, we configure a *watchdog timer* at all components (cars and ICS) that triggers a switch to manual mode.

3. Related Work

There exist numerous component models with different properties and application domains [11][12]. In this section, we position our contribution with respect to the most relevant such models used in embedded systems. In particular, we discuss their support for dynamic architectures and open-endedness (i.e., allowing adaptivity and reconfiguration), real-time behavior (i.e., providing deterministic running times), and real-time analysis (i.e., providing techniques or tools for timing analysis and guarantees) as required by our ICS setting.

Closest to DEECo [4] in its ability to deal with open-ended and dynamic systems are Helena [5] and jRESP [13]. They both build on the same concept of autonomic component ensembles. However, neither of them provide low footprint and real-time properties needed for embedded systems. By excluding any real-time properties, they also naturally miss any methods for real-time analysis.

The most prominent example of component models targeting embedded systems is certainly AUTOSAR [2], which is of common use in the automotive industry. AUTOSAR serves as a specification for different layers – i.e., application software, runtime environment (RTE) and basic software (BSW) – and inherently accounts for distribution (the RTE provides location transparency to application components). However, in AUTOSAR components are statically linked

with BSW and the RTE, which does not support open-endedness and dynamic architectures. AUTOSAR itself does not provide timing analysis, but it has been enriched by TIMing MOdel (TIMMO) [14], which builds on the Timing Augmented Description Language (TADL).

BlueArx [15] is a component framework backed by Bosch targeting especially the automotive domain. Similar to AUTOSAR, it offers component-based architecture with internal variability (achieved by modes) and connects real-time specification to it. It allows using existing tools for timing analysis (e.g., AbsInt aiT for WCET estimation). Components are assembled at compile time which makes BlueArx neither open-ended nor truly dynamic (the dynamism is restricted to switching among pre-defined modes).

Koala [16] together with its de facto successor ROBOCOP [17] constitutes an approach to component-based design of software for consumer electronics by Philips. Their primary purpose is facilitating the product-line engineering of the software. They do not specifically target real-time attributes or analysis. Contrary to Koala, ROBOCOP allows for component binding at runtime, which theoretically creates a basis for open-ended systems. However, it lacks the features for system self-organization as provided, for instance, by DEECo concept of ensembles.

IEC 61499 is a standard developed by IEC (International Electrotechnical Commission) for development of automation and control systems [18]. It provides function blocks and their interconnections via defined ports - basically featuring the component-based concepts. IEC 61499 has a static architecture and does not provide any means to specify and analyze real-time properties. Extensions, however, have been made (e.g., in [19]), which already permits modeling real-time attributes, but is still static.

Think [20, 21] and MIND [22] are C-based implementation of Fractal [23] targeted for embedded and real-time systems. As they rely on Fractals API for runtime architecture introspection and binding, they provide the basic infrastructure for dynamic and open-ended architectures. However, similar to ROBOCOP, they lack the features for automated system self-organization as provided by DEECos concept of ensembles. They do not address real-time properties or analysis.

SOFA-HI [24] is an extension of the SOFA component model [25] to embedded and real-time systems. It features passive and active components – the active component are associated with real-time properties that drive their scheduling; however, no particular real-time analysis technique has been elaborated for SOFA-HI so far. The architecture of SOFA-HI is further static (limited only to switching among pre-defined modes).

OpenCOM [26] is a component model that features reflective capabilities providing the basic means for dynamic architectures. However, this component model does neither target real-time systems, nor does it allow any related analysis. Another component model named PIN [27] describes static architectures supporting real-time attributes and the related analysis – in particular, prediction of average latency and formal verification of temporal safety.

The design-level approaches (not necessarily bound to a particular runtime framework) are represented by AADL and UML-based approaches – most notably UML and SysML together with MARTE. AADL (Architecture Analysis and Design language) [28] is a widely used model for describing the software and hardware architecture of an embedded system. In supporting timing analysis, it relies on Real-Time Calculus (RTC) [29], which is a formalism that allows for system-level performance analysis of stream-processing systems constrained by hard real-time requirements. Essentially RTC models are extracted from AADL and subsequently the RTC tools can be employed. The architecture, however, is assumed to be static.

UML [30] represents a typical approach for modeling architecture. Though not exactly tailored towards real-time embedded systems, its ability to capture structures and behavior applies even in this context. For real-time properties, it is typically connected with MARTE – a UML

profile for real-time systems. MARTE comes with a number of approaches and tools that allow estimating a system's timing behavior (e.g., [31]).

In the course of bringing UML to the embedded community, OMG defined SysML [32], which is a close sibling of UML, generally adopting most of its diagrams and extending them with the ability to model physical structures. SysML again integrates with MARTE [33] to enable modeling non-functional properties such as power consumption, performance and timing and their estimation [34]. Though rather mature, they again assume static component architectures, which effectively hinders their direct application in open-ended system.

Other component models such as PECOS [35], COMDES-II [36], Rubus[37], SaveCCM [38], ProCom [39], MyCCM-HI [40], and PCM[41] support real-time properties and analysis, but do not account for dynamic architectures nor open-ended systems and, hence, are not suitable for our ICS case study. On the other hand, CIAO [42] supports dynamic architectures and real-time properties. However, it does not provide any support for open-endedness nor for real-time analysis.

Looking into models that do not specifically target embedded systems, timing analysis enabled at the model level is also supported by the BIP (Behavior, Interaction, Priority) framework [43]. BIP supports real-time aspects by using *timed components*, which allow for timing properties being specified using *timed variables* and *transitions*. Those are accounted for during the validation within the real-time engine implementing the operational semantics of BIP models. The composition is, however, static not providing any support for open-ended systems.

An interesting (not component-based) approach is proposed by Etzien et al. in [44][45]. The authors describe a modeling method for evolutionary distributed systems using the concept of System of Systems (SoS). In order to capture both static and dynamic properties of the developed system, they use the contract paradigm for specifying legal configurations of a SoS and to describe architectural changes during runtime. In [46], the authors extend their work by providing a method for schedulability analysis. They based their technique on both analytical and model checking methods, which combined with the SoS contracts provide for a compositional and scalable solution. However, in order to perform the analysis of [46], one needs to deal with a full-fledged implementation of the system, from which necessary parameters are extracted to construct a state machine for analysis. Such method is rather suitable for verification and validation of an existing implementation. In our case, the proposed analysis is intended to be used at an early design phase where mostly system requirements are known (see Figure 1). Moreover, our analysis addresses CPS with a strong connection to the environment – see Section 6, whereas [46] focuses on more general-purpose systems.

Table 1 presents a summary comparing the component models/approaches mentioned above with respect to the three key features required for implementing our ICS case study.

4. Modeling with DEECo

As stated above, we make use of the DEECo component model [4]. DEECo describes the architecture of a CPS by means of components (i.e., encapsulated well-defined active entities, which perform sensing, computation and actuation) and so called ensembles, which are dynamically established groups of components that cooperate to achieve a particular goal. DEECo further provides a special requirements engineering method and traceability of requirements to components and ensembles – for further details, we refer to [7].

Component Model/Approach	Dynamic Architectures/ Open-Endedness	Real-Time Behavior	Real-Time Analysis
DEECo [4]	✓✓	✓	✓
Helena [5]	✓✓	✗	✗
jRESP [13]	✓✓	✗	✗
AUTOSAR/TIMMO [2][14]	✗	✓	✓
BlueArx [15]	✗	✓	✓
Koala [16]	✗	✗	✗
ROBOCOP [17]	✓	✗	✗
IEC 61499 [18][19]	✗	✓	✓
Think/MIND [21][22]	✓	✗	✗
SOFA-HI [24]	✗	✓	✗
OpenCOM [26]	✓	✗	✗
PIN [27]	✗	✓	✓
AADL [28]	✗	✓	✓
UML/SysML with MARTE [31][33]	✗	✓	✓
PECOS [35]	✗	✓	✓
COMDES-II [36]	✗	✓	✓
Rubus/SaveCCM [37][38]	✗	✓	✓
ProCom [39]	✗	✓	✓
MyCCM-HI [40]	✗	✓	✓
PCM [41]	✗	✓	✓
CIAO [42]	✓	✓	✗
BIP [43]	✗	✓	✓
Etzien et al. [44][45][46]	✓	✓	✓

Table 1: Support for dynamic architectures/open-endedness (two check marks mean that special constructs are provided to describe how a system should self-organize in open-ended settings), real-time behavior, and real-time analysis

4.1. Architecture Modeling

Components. To illustrate the principles behind DEECo, Listing 1 depicts a component using a DSL (Domain-Specific Language) description.[1] In DEECo, each component consists of *knowledge* – see lines 8-16 – reflecting its current state. Knowledge is expressed by attributes organized into hierarchical data structures. Access to one or more such attributes of a component is performed through interfaces – see lines 1-5 – that are featured by the component.

In addition, each component has a set of processes (essentially real-time tasks) that manipulate its knowledge. A process is characterized by a function (e.g., lines 20-22), whose parameter list consists of knowledge attributes. DEECo's *runtime environment* manages the release of processes and takes care of knowledge retrieval before a process is executed and knowledge update – also referred to as knowledge exchange – when a process finishes executing. Each of the component's processes is executed in isolation meaning that it is not supposed to communicate with other processes (either from the same or a different component) in any other way than via the component's knowledge.

[1]Note that this DSL specification serves for demonstration only. Later we discuss how to derive a C++ implementation from this specification, which can then be used on embedded devices.

```
1  interface MovingUnit:
2      id, time, crossingId, crossingDistance, crossingDirection, speed, privileged, mode
3
4  interface MovingUnitAggregator:
5      id, time, vehicles, speeds, mode
6
7  component Vehicle features MovingUnit
8      knowledge:
9          id: 42,
10         time: 1440691842456 ms,
11         crossingId: 12
12         crossingDistance: 35 m,
13         crossingDirection: South−West,
14         speed: 50 Km/h,
15         privileged: FALSE,
16         mode: AUTOMATIC,
17         ...
18     process UpdateSpeed:
19             in speed
20         function:
21             Actuators.setSpeed(speed);
22         scheduling: periodic( 5 ms )
23     ...
24
25 component ICS features MovingUnitAggregator
26     knowledge:
27         id: 12,
28         time: 1440691842458 ms,
29         vehicles: [...],
30         speeds [...]
31         privileged: [...],
32         mode: AUTOMATIC,
33         ...
34     process findPrivilegedVehicles:
35             in vehicles, inout privileged
36         function:
37             for (v : vehicles)
38                 if (v.privileged)
39                     privileged.add(v)
40         scheduling: triggered( changed(movingUnits) )
41     ...
```

Listing 1: DEECo component definitions based on a DSL

A process can be executed in response to a timer event (i.e., periodic execution) or as a reaction to a change in one of its attributes. In our example, the *Vehicle* component has a process that sets/updates the speed of the car or vehicle. This is repeated periodically every 5 ms (see line 22). As another example the ICS process, shown in Listing 1, determines whether there are privileged vehicles in its region of influence (lines 34-40) and is executed whenever the number of vehicles changes (line 40). Once processes are released (by DEECo's runtime environment), these are handed over to the platform's operating system (OS), which is responsible for scheduling them according to a desired policy – see Section 4.2.

```
1  ensemble UpdateMovingUnitInformation:
2      coordinator: MovingUnitAggregator
3      member: MovingUnit
4      membership:
5          coordinator.id = member.crossingId, member.crossingDistance < 50 m
6      knowledge exchange:
7          coordinator.movingUnits.add({member})
8      scheduling: periodic( p_ens,i )
```

Listing 2: A DSL example of an ensemble definition.

Ensembles. An ensemble in DEECo defines a semantic connector between components and constitutes their *composition*. The composition in DEECo is *flat* and occurs implicitly by com-

ponents dynamically joining an ensemble at runtime. When specifying an ensemble, prospective components are described by roles. One component in the ensemble has a *coordinator's* role, whereas the remaining components are *members* of the ensemble.

The roles are defined by the interfaces – in our example, *MovingUnit* and *MovingUnitAggregator* – which are matched at runtime to the actual components (i.e., their knowledge) for a structural coincidence. Later, those components with matching interfaces are considered for the ensemble evaluation process, which is composed of two steps. The first step involves checking the *membership condition*, which is expressed as a logic predicate formulated upon coordinator's and member's attributes. This determines whether two components (a coordinator and a member component) should form an ensemble. The second step depends on the results of the membership condition check and consists of exchanging attribute values between coordinator and member according to the description given in the *knowledge exchange* specification.

In the example in Listing 2, the coordinator role is determined by the interface definition *MovingUnitAggregator* and the member role by *MovingUnit*. This way, during the ensemble evaluation, only components featuring appropriate interfaces will be considered. The membership condition further constraints the number of ensemble members to those, which are located no more than 50 m from the coordinator's location. Then, according to the knowledge exchange description, the coordinator's *movingUnits* attribute is updated with information about all components that fulfill the membership condition (which is checked every $p_{ens,i}$ time units – see line 8 – with i being an index representing the component). This way, the ICS is aware only of those vehicles, which are currently in its close proximity.

```
1  namespace VehicleComponent {
2  struct Knowledge: CDEECO::Knowledge {
3      VehicleId id;
4      Time time;
5      CrossingId crossingId;
6      Distance crossingDistance;
7      Direction crossingDirection;
8      Speed speed;
9      bool privileged;
10     Mode mode;
11     ...
12 };
13
14 class UpdateDistance: public
15         CDEECO::PeriodicTask<Knowledge, Distance> {
16     UpdateDistance(auto &component);
17     Distance run(const Knowledge in);
18 };
19
20 class SetSpeed: public CDEECO::PeriodicTask<Knowledge, void> {
21     SetSpeed(auto &component);
22     void run(const Knowledge in);
23 };
24
25 class Component: public CDEECO::Component<Knowledge> {
26     static const CDEECO::Type Type = 0x00000001;
27
28     UpdateDistance updateDistance = updateDistance(*this);
29     SetSpeed setSpeed = SetSpeed(*this);
30     ...
31
32     Component(CDEECO::Broadcaster &broadcaster,
33         const CDEECO::Id id,
34         bool remotelyOperable);
35 };
36 }
```

Listing 3: A C++ example of Vehicle component.

DEECo's deterministic semantics. DEECo components are autonomous and rely only on data that is present in their knowledge. As mentioned before, any interaction of a component with other components is realized by ensembles, which is *externalized* from the component itself. This property of DEECo's component model suits very well to both the design and the implementation of distributed adaptive systems, since all technical aspects related to communication between remote components are abstracted away from the design phase and left for the runtime environment to deal with them.

Technically, the runtime environment periodically *propagates* ensemble-relevant knowledge to all other components or nodes in the system – note that gossip-based algorithms [47] might be used for this purpose. In our case study, ensemble-relevant data are the car's distance to the crossing, its speed, and its intended direction, etc. This is used to evaluate whether cars are heading in the direction of the crossing or not.

Each node then keeps relevant reference knowledge from all other nodes from which it has received data. In other words, components or nodes join (and leave) the system in an implicit manner without performing any *handshaking*. Since ensemble-relevant information is present at all nodes, DEECo's runtime environment performs a local evaluation of an ensemble membership condition. If this holds true, the local reference knowledge of the remote components involved is used for the *knowledge exchange* process.

In this way, DEECo's semantics separates *decision taking* (i.e., ensemble evaluation and its eventual knowledge exchange) from *information sharing* (i.e., knowledge propagation) at the components. Since a DEECo component takes decisions based on locally available data, it does not need to synchronize with other components in the system. On the one hand, this has the advantage of high flexibility and adaptability. On the other hand, clearly, local data might get outdated at the different nodes, which needs to be analyzed carefully as illustrated in Section 5.

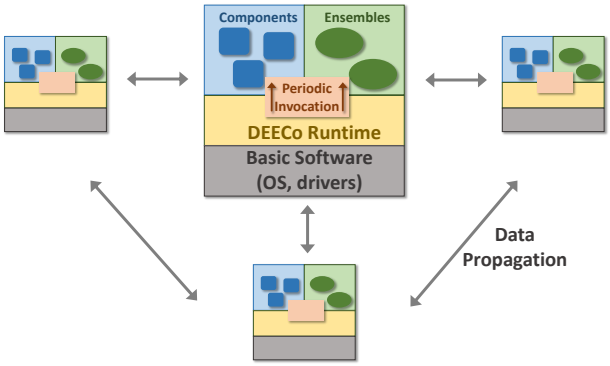

Figure 3: DEECo distributed deployment.

4.2. Implementation and deployment

The implementation and distributed deployment of DEECo systems are supported by the CDEECo framework. This framework maps DEECo concepts to C++ and constitutes our runtime environment (taking care of periodically propagating knowledge, performing ensemble evaluations, performing knowledge exchange if applicable, etc.). As depicted in Figure 3, CDEECo

relies on an OS providing hardware abstraction and other services. Clearly, this OS needs to support real-time behavior, i.e., real-time scheduling, interrupt handling, etc., to be used in safety-critical applications.

With respect to CDEECo's implementation in C++, components and processes are handled as *classes*, while knowledge is treated as a static data structure (with a fixed size in bytes). Thus it is possible to operate on knowledge within bounded time. Moreover, it is easy to fragment knowledge to fit into network packets, also within bounded time.

```
1  namespace VehicleInfoAndSpeedExchange {
2  typedef CDEECO::Ensemble< ICS::Knowledge, Vehicle::Knowledge*,
3          Vehicle::Knowledge, Speed> EnsembleType;
4
5  class Ensemble: EnsembleType {
6      Ensemble(CDEECO::Component<ICS::Knowledge> &coord,
7          CDEECO::KnowledgeLibrary<Vehicle::Knowledge> &lib);
8
9      Ensemble(CDEECO::Component<Vehicle::Knowledge> &mbr,
10         CDEECO::KnowledgeLibrary<ICS::Knowledge> &lib);
11
12      bool isMember(const CDEECO::Id coordId,
13          const ICS::Knowledge coordKnowledge,
14          const CDEECO::Id memberId,
15          const Vehicle::Knowledge memberKnowledge);
16
17     Vehicle::Knowledge* memberToCoordMap(
18         const ICS::Knowledge coord,
19         const CDEECO::Id memberId,
20         const Vehicle::Knowledge memberKnowledge);
21
22     Speed coordToMemberMap(
23         const Vehicle::Knowledge member,
24         const CDEECO::Id coordId,
25         const ICS::Knowledge coordKnowledge);
26 };
27 }
```

Listing 4: A C++ example of vehicle-crossroad ensemble

A component is represented by a class which inherits *basic behavior* from a template with *knowledge* as an argument. Similarly, processes are represented by classes inheriting basic behavior from a template, depending on the process type, while accepting input and output types as template arguments. The process's code is contained in a *virtual method* in the base class. The component class, process classes, and the knowledge data structure can be wrapped in a *namespace* to improve code readability as shown in Listing 3.

Similarly to components, ensembles are also implemented using classes inheriting basic behavior from a template. The membership method and a pair of knowledge mapping methods are realized as virtual methods in the base class. In case of ensembles, template arguments are more complex, since the data type have to capture the coordinator's and the member's input/output *knowledge types*. In order to simplify the code, the complex type of the ensemble can be defined using typedef and wrapped together with ensemble class in a *namespace* as displayed in Listing 4.

CDEECo's sources are located on GitHub[2]. So far, we have been using FreeRTOS[3] as an OS to schedule DEECo processes on the corresponding nodes. As mentioned above, CDEECo periodically propagates knowledge over available communication channels, in our case study, a

[2] `https://github.com/d3scomp/CDEECo`
[3] `http://freertos.org/`

VANet. To this end, it periodically broadcasts *binary patches* covering the whole corresponding component's knowledge. These are bulks of binary data with offset, size and source component ID (i.e., an identification code). Patches are then used to update knowledge on the receiver component when no packet is lost. Otherwise the knowledge can be just partially updated.

This solution was chosen to maximize knowledge propagation, while the consistency can be achieved by storing dependent data in single packets. Knowledge that has been successfully received from remote components is stored in a so-called *knowledge library*. A knowledge library is a data structure holding predefined number of remote knowledge data sets; when an ensemble is evaluated true, a knowledge exchange is triggered copying (locally available) data from the knowledge library to the local component's memory space.

5. DEECo's Closed-Loop Reaction Time

In this section, we analyze DEECo's closed-loop reaction time in the worst case. This is defined as the maximum delay that it takes a DEECo-based system to react to changes in the environment. The term *closed-loop* reflects the fact that DEECo components interact with one another.

For example, if a component A experiences a change in its internal states, e.g., due to one or more physical variables measured by its sensors, this will take some time to reach another component B – connected by ensembles – in the system. Similarly, component B's reaction to the change in component A will take some additional time to reach back component A. The sum of these two times is the closed-loop delay between A to B. In other words, component A and B form a loop.

For ease of exposition, we first make use of our case study and then generalize our results to make them independent of the application. In our case study, knowledge needs to be exchanged from a car to the ICS and from the ICS back to the car for the system to work as expected. However, knowledge exchange happens based on *local data* when the corresponding ensemble condition is evaluated to true at both the ICS and the car nodes separately.

As discussed above, knowledge is propagated (from the car to the ICS and vice versa) and the ensemble membership check is performed (at the car and at the ICS) on a periodic basis. Let us denote by $\hat{p}_{pro}$ the maximum period with which knowledge is propagated by any node in the system. Similarly, let $\hat{p}_{ens}$ be the maximum period with which ensembles are evaluated at any node in the system. That is:

$$\hat{p}_{pro} = \max_{\forall i} \left(p_{pro,i} \right),$$
$$\hat{p}_{ens} = \max_{\forall i} \left(p_{ens,i} \right),$$

where $p_{pro,i}$ and $p_{ens,i}$ are the knowledge propagation and the ensemble evaluation period of a node i, respectively, with i being an index that identifies the corresponding component/node.

In order to eliminate the need for synchronization and handshaking between components, these two processes in DEECo are not synchronized with one another and, hence, we have the following conditions in the worst case:

i) A car propagates its knowledge to the ICS immediately after a membership evaluation has been performed at the ICS. As a result, data is received $\hat{p}_{ens}$ time later at the ICS, when the next membership evaluation is performed.

ii) In a similar manner, the ICS propagates its knowledge to the corresponding car just after a membership evaluation has been performed at the car. As a result, data is received $\hat{p}_{ens}$ time later at the car too.

iii) The knowledge of the car changes immediately after knowledge has been propagated to the ICS. As a result, the *current* knowledge is propagated with a delay $\hat{p}_{pro}$ from the car to the ICS, when a new propagation is performed.

iv) The ICS's knowledge changes immediately after knowledge has been propagated to the car. Hence, the *current* knowledge is not propagated until a new propagation is started $\hat{p}_{pro}$ time later.

As a result, in the worst case, we have a delay due to the asynchronous nature of the DEECo framework which is given by the following expression:

$$2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens}. \tag{1}$$

In addition, there is also a process running at the ICS which computes the speed for the car that guarantees no collisions at the current traffic situation. This process is triggered when a knowledge exchange is executed at the ICS (i.e., when the ensemble is evaluated to true between the car and the ICS). We denote by r_{ICS} the worst-case response time (WCRT) of this process. Analogously, there is a process running at the car, which applies the new speed values to the *physical* car. This process is also triggered when a knowledge exchange happens at the car and its WCRT is r_{car}.

As a result, the worst-case delay D_{max} for a closed-loop reaction in DEECo, i.e., a reaction of a car to an input from the ICS computed based on the car's current knowledge, is given by the following equation also illustrated in Figure 4:

$$D_{max} = 2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens} + c_{ICS} + c_{car} + r_{ICS} + r_{car}, \tag{2}$$

where c_{car} is the communication delay from the car to the ICS and c_{ICS} is the communication delay from the ICS to the car.

Since there is interference by other messages (from other cars), c_{car} is the maximum possible communication delay in the network. However, from the ICS to the car, there is no interference – assuming a full-duplex communication channel – and the communication delay c_{ICS} is equal to the transmission time, since the ICS does not compete for accessing the network.

The term $2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens}$ in Equation (2) is clearly intrinsic to DEECo and does not depend on the application, but rather on how components are configured. In addition, $c_{ICS} + c_{car}$ is the total communication delay between the ICS and a car, whereas $r_{ICS} + r_{car}$ is the delay due to computation at the ICS and at the car. As a result, DEECo's closed-loop delay in the worst-case can be generalized, i.e., made independent of the application under consideration, as follows:

$$\bar{D}_{max} = 2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens} + C_{max} + R_{max}, \tag{3}$$

where C_{max} is the sum of the worst-case communication delay between any two DEECo components and R_{max} is the sum of the WCRTs of computation processes involved at the corresponding components or nodes.

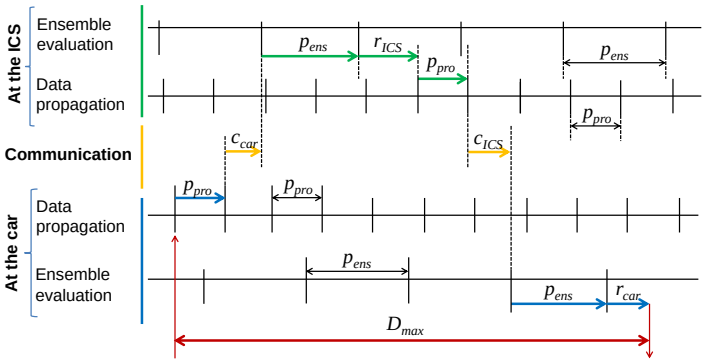

Figure 4: Composition of DEECo's closed-loop delay D_{max}: In the worst case, data may change immediately after knowledge has been propagated at the car. This data may also arrive after an ensemble evaluation has been performed at the ICS. In addition, computation at the ICS may finish immediately after knowledge propagation and the resulting data then reaches the car just after the end of an ensemble evaluation.

6. Real-Time Analysis

The purpose of performing a real-time analysis is to guarantee that timing constraints can be met by the system, which is required in safety-critical applications. In the case of DEECo, this boils down to checking that $\bar{D}_{max}$ as per Equation (3) is below a given time upper bound, which stems from physical processes involved and needs to be known to or obtained by the designer. In turn, $\bar{D}_{max}$ depends on C_{max} and R_{max}, i.e., on the delays incurred by communication and computation processes involved – which are closely related to the technologies and techniques used for implementing the system.

All in all, our real-time analysis consists of the following steps that we illustrate next in the context of our case study: i) obtaining the worst-case computation delay, ii) obtaining the worst-case communication delay, iii) determining system constraints, and iv) obtaining a feasible DEECo configuration.

6.1. Obtaining the worst-case computation delay

As discussed previously, CDEECo – DEECo's runtime environment – is executed on top of an OS at each node in the system. Among others, CDEECo is in charge of releasing component's processes as specified in the component description. Clearly, to be used in safety-critical applications, CDEECo relies on specific technologies that make real-time scheduling and real-time communication possible. In particular, the OS needs to support real-time scheduling; otherwise, it will not be possible to guarantee real-time behavior.

The techniques for schedulability analysis, i.e., testing whether processes meet their deadlines at the different nodes, strongly depend on the scheduling algorithm used. As mentioned above, CDEECo makes use of FreeRTOS, which supports fixed-priority scheduling and allows for the *rate monotonic* policy [48]. That is, processes are given fixed priorities according to the following rule: The shorter a process's period is, the higher the priority assigned to it.

Let us denote by $\mathbf{T}$ the set of processes on a given node. Further, τ_i is a process that belongs to $\mathbf{T}$ where e_i denotes its worst-case execution time (WCET) and p_i denotes its period of repetition. For $\mathbf{T}$ to be schedulable, the following has to hold for each τ_i and $1 \leq i \leq |\mathbf{T}|$:

$$\sum_{\forall \tau_j \in \widehat{\mathbf{T}}} \left\lceil \frac{p_i}{p_j} \right\rceil e_j \leq p_i, \tag{4}$$

where $|\mathbf{T}|$ is the number of elements in $\mathbf{T}$ and $\widehat{\mathbf{T}}$ denotes the subset of processes from $\mathbf{T}$ which have a higher priority than the corresponding τ_i.

This expression means that for each process τ_i to be schedulable (and, hence, for $\mathbf{T}$ to be schedulable), the sum of all executions of higher-priority processes in a time interval equal to p_i plus its own execution e_i should be less than its deadline p_i. Note that Equation (4) is sufficient but not necessary. A sufficient and necessary test can be achieved by response time analysis [49]; however, the sufficient test of Equation (4) is enough for the purpose of this paper. Now, since the following holds:

$$\sum_{\forall \tau_j \in \widehat{\mathbf{T}}} \left\lceil \frac{p_i}{p_j} \right\rceil e_j \quad \leq \quad \sum_{\forall \tau_j \in \widehat{\mathbf{T}}} \left(\frac{p_i}{p_j} + 1 \right) e_j,$$

we can reshape Equation (4) to:

$$\sum_{\forall \tau_j \in \widehat{\mathbf{T}}} \frac{e_j}{p_j} + \frac{\sum_{\forall \tau_j \in \widehat{\mathbf{T}}} (e_j)}{p_i} \leq 1. \tag{5}$$

Clearly, if Equation (5) holds, Equation (4) will also hold. However, Equation (5) is easier to compute and operate with.

In our case study, to obtain $R_{max} = r_{ICS} + r_{car}$, let us denote by $\mathbf{T}_{ICS}$ the set of computation processes at the ICS. Further, we assume that a process τ_i is the one involved in the closed-loop delay, i.e., the one computing the new speed for a given car. Considering that $\widehat{\mathbf{T}}_{ICS}$ is the subset with higher- or equal priority processes than τ_i at the ICS, we can compute r_{ICS} as follows:

$$r_{ICS} = \left(\sum_{\forall \tau_j \in \widehat{\mathbf{T}}_{ICS}} \frac{e_j}{p_j} \right) p_i + \sum_{\forall \tau_j \in \widehat{\mathbf{T}}_{ICS}} e_j. \tag{6}$$

Analogously, at the car, we can obtain r_{car} as follows:

$$r_{car} = \left(\sum_{\forall \tau_j \in \widehat{\mathbf{T}}_{car}} \frac{e_j}{p_j} \right) p_i + \sum_{\forall \tau_j \in \widehat{\mathbf{T}}_{car}} e_j. \tag{7}$$

6.2. Obtaining the worst-case communication delay

Similar to computation delay, communication delay strongly depends on the underlying technologies and techniques used. Since VANets are usually based on wireless communication [50], we assume that Ethernet IEEE 802.1Q is the underlying protocol, which provides mechanisms to prioritize *messages* [51]. In general, we will normally have a number of access points (AP) which are connected to a full-duplex switch via Ethernet.

In this section, we consider that communication to the AP is collision-free as per some VANet scheme – later we remove this assumption to analyze the effect of packet loss. For example, space division multiple access (SDMA) has been proposed [52, 53], which guarantees a collision-free

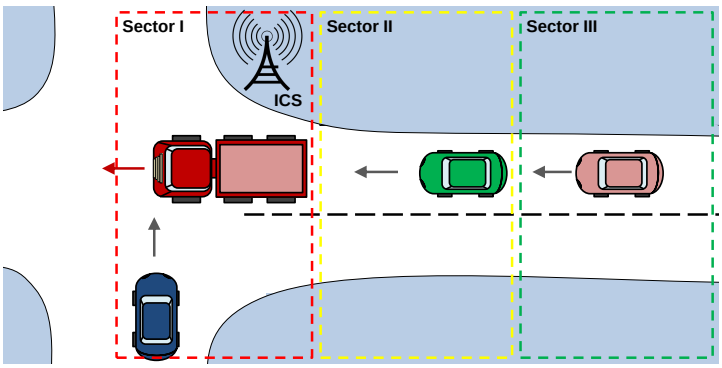

Figure 5: Priorities are given according to the proximity to the intersection

communication in busy intersections by dividing a road into sectors and assigning different time slots to each such sector. Other solutions combine special antennas with also a TDMA (time division multiple access) scheme to reduce packet loss [54, 55].

Now, assuming that wireless network provides 100 Mbps and that messages are at most 1 Kbit (1024 bits), then the transmission time c_W on the wireless network is at most – considering a 144-bit protocol overhead:

$$c_W = \frac{1024 + 144}{100\,\text{Mbps}} = 11.68\,\mu\text{s}.$$

However, the switch then sends messages to the ICS according to their priorities. Let us consider that the ICS's region of influence is divided into sectors with different priorities – see Figure 5. Cars that are in the first sector (e.g., within 10 m from the intersection) have higher priority than cars in the second sector (e.g., from 10 m to 20 m) and so on. At a given point in time, if a car is in more than one sector simultaneously, it will be assigned the highest priority among those sectors. The switch then sends messages to the ICS according to these priorities.

Let us now analyze the communication segment between the switch and the ICS. To this end, let $\mathbf{M}$ denote the set of all messages being sent to the ICS over the switch. Further, m_i denotes one such message in $\mathbf{M}$ where c_i is its transmission time – note that c_i is constant for a given i which results from the amount of bits to be sent and the bandwidth of the communication channel – and z_i denotes the minimum inter-arrival time between two consecutive such messages. The deadline of a message is also given by z_i.

Generally, for all messages in $\mathbf{M}$ to meet their deadlines, the following has to hold for $1 \leq i \leq |\mathbf{M}|$, where $|\mathbf{M}|$ is the number of elements in $\mathbf{M}$:

$$b_i + \sum_{\forall m_j \in \widehat{\mathbf{M}}} \left\lceil \frac{z_i}{z_j} \right\rceil c_j \leq z_i, \tag{8}$$

where $\widehat{\mathbf{M}}$ is the subset of $\mathbf{M}$ with higher- or equal priority messages than m_i and b_i denotes *blocking time* on the communication channel. That is, whenever a message needs to be sent, a lower-priority message might eventually be using the communication channel. Since this lower-priority message cannot be interrupted, there is a blocking time on the bus. Clearly, in worst case, b_i is given by the maximum transmission time among all lower-priority messages:

$$b_i = \max_{l=i}^{|\mathbf{M}|}(c_l). \tag{9}$$

Considering that $\sum_{\forall m_j \in \widehat{\mathbf{M}}} \left\lceil \frac{z_i}{z_j} \right\rceil \leq \sum_{\forall m_j \in \widehat{\mathbf{M}}} \left(\frac{z_i}{z_j} + 1 \right)$ holds, we can remove the ceiling function and approximate Equation (8) as shown below:

$$\sum_{\forall m_j \in \widehat{\mathbf{M}}} \frac{c_j}{z_j} + \frac{b_i + \sum_{\forall m_j \in \widehat{\mathbf{M}}}(c_j)}{z_i} \leq 1. \tag{10}$$

To demonstrate this in our case study, let us assume that the Ethernet link between the switch and the ICS has a bandwidth of 1 Gbps. If messages have a length of at most 1 Kbit (1024 bits), and the protocol overhead is of 144 bits, we have that the transmission time c_i of a message m_i is given by $\check{c}_E$:

$$\check{c}_E = \frac{1024 + 144}{1\,\text{Gbits}} = 1.168\,\mu\text{s}.$$

Further, with help of Equation (10), we can compute the transmission time on the Ethernet link taking contention by higher- and equal priority messages into account, which we denote by $\hat{c}_E$:

$$\hat{c}_E = \left(\sum_{\forall m_j \in \widehat{\mathbf{M}}} \frac{c_j}{z_j} \right) z_i + b_i + \sum_{\forall m_j \in \widehat{\mathbf{M}}} c_j. \tag{11}$$

In addition to the transmission time, there is always a delay at the AP and at switches in Ethernet – denoted by e_{AP} and e_{SW} respectively, which accounts for buffering and routing tasks. This is typically in the order of $2\,\mu$s.

$C_{max} = c_{ICS} + c_{car}$ can now be obtained. To this end, recall that we have two Ethernet links: one from the AP to the switch and another from the switch to the ICS. The ICS does not suffer from contention at the communication channel, since the connection from and to the APs is assumed to be full duplex. As a result, c_{ICS} is given by:

$$c_{ICS} = 2 \times \check{c}_E + c_W + e_{SW} + e_{AP}. \tag{12}$$

On the other hand, cars share the communication channel and, hence, they may have contention at the communication channel leading to a c_{car} as follows:

$$c_{car} = 2 \times \hat{c}_E + c_W + e_{SW} + e_{AP}. \tag{13}$$

6.3. Determining system constraints

Timing constraints are clearly derived from the application. In our case, we consider that a car needs to be provided with new speed values at every single meter of its trajectory (taking vehicle dynamics into account such inertia, braking distance, etc.). If a car's speed is at maximum 50 Km/h (assuming an urban scenario), we need to compute the time $t_{1\text{m}}$ that it needs to cover 1 m of its trajectory:

$$t_{1\text{m}} = \frac{1\,\text{m} \times 3600\,\text{s/h}}{50 \cdot 10^3\,\text{m/h}} = 72\,\text{ms}. \tag{14}$$

On the other hand, the computation and communication overhead depend on the number of components, in particular, cars/vehicles in the system, which is the second constraint from the

application. Clearly, the more cars enter the ICS's region of influence, the more computation and communication overhead there will be. To compute the maximum possible number of cars at the crossing, let us assume that a car is at least 2 m long and that there is a least a 1 m distance between any two cars. As a result of this, in the worst possible case, the number of cars n approaching the intersection from all directions is given by the following equation:

$$n = 4 \times \left\lceil \frac{50\,\mathrm{m}}{3\,\mathrm{m}} \right\rceil = 68. \tag{15}$$

We can use Equation (15) to configure $\hat{p}_{ens}$ and $\hat{p}_{pro}$ in the next section and the timing constraint as per Equation (14) to perform a feasibility analysis as discussed later.

6.4. Obtaining a feasible DEECo configuration

There will be at most 68 different ensemble instances (between the ICS and each of the cars) at the ICS – see Equation (15). In addition, there will be 68 processes to compute new speed values for each car. Since the ensemble membership check triggers a knowledge exchange – recall that knowledge exchange is based on locally available data and that knowledge/data propagation (from the ICS to the cars and vice versa) is a separate and asynchronous process – when evaluated true, we can assume that in the worst case all 68 ensemble processes trigger their corresponding computation processes simultaneously. In addition, there will be one knowledge propagation process for the ICS[4].

Assuming that all processes have a WCET $e_i = 25\mu$ s (note that most these processes consist in checking logic conditions, assigning pointers to given memory spaces, etc., or are simple computations), we can use Equation (5) applied to the ICS as follows:

$$\frac{0.025\,\mathrm{ms}}{\check{p}_{pro}} + 68 \cdot \frac{2 \times 0.025\,\mathrm{ms}}{\check{p}_{ens}} + \frac{(0.025\,\mathrm{ms} + 68 \cdot (2 \times 0.025\,\mathrm{ms})}{\check{p}_{ens}} \leq 1, \tag{16}$$

where $\check{p}_{pro}$ and $\check{p}_{ens}$ are the minimum periods with which knowledge is propagated and with which ensembles are evaluated in the system. That is:

$$\check{p}_{pro} = \min_{\forall i} \left(p_{pro,i} \right),$$
$$\check{p}_{ens} = \min_{\forall i} \left(p_{ens,i} \right).$$

Note that if Equation (16) holds for $\check{p}_{pro}$ and $\check{p}_{ens}$, it will also hold for any $p_{pro,i}$ and $p_{ens,i}$. We obtain the following value for $\check{p}_{ens}$ assuming $2 \times \check{p}_{pro} = \check{p}_{ens}$, i.e., that knowledge propagation is done twice as frequently as any ensemble membership check[5]:

$$\check{p}_{ens} \geq 6.88\,\mathrm{ms},$$

and, hence, $\check{p}_{pro}$ has to be greater or equal to 3.44ms. Note that there cannot be a $p_{ens,i}$ that is less than $\check{p}_{ens}$. Similarly, $p_{pro,i}$ is bounded from below by $\check{p}_{pro}$. Otherwise, Equation (16) will not hold.

[4]Note that the knowledge propagation from cars does not produce any overhead at the ICS, but at the respective cars.

[5]This is a design decision that needs to be taken. In general, since ensemble membership checks rely on local knowledge, it is meaningful that knowledge be updated as often as necessary to guarantee desired functionality.

On the other hand, the upper bounds $\hat{p}_{ens}$ and $\hat{p}_{pro,i}$ need to fulfill the system's feasibility condition. That is, DEECo's closed-loop delay must be at most equal to the timing constraint $t_{1\mathrm{m}}$:

$$D_{max} \leq t_{1\mathrm{m}}. \tag{17}$$

D_{max} is DEECo's closed-loop delay for our case study as per Equation (2), i.e., where $R_{max} = r_{ICS} + r_{car}$ and $C_{max} = c_{ICS} + c_{car}$ in general expression $\bar{D}_{max}$ as given in Equation (3).

We choose $\hat{p}_{ens} = 14\,\mathrm{ms}$ – twice as much as $\check{p}_{ens}$ – and hence $\hat{p}_{pro} = 7\,\mathrm{ms}$, i.e., we again have $2 \times \hat{p}_{pro} = \hat{p}_{ens}$. With these values of $\hat{p}_{ens}$ and $\hat{p}_{pro}$, we verify next whether Equation (17) can be met. If not, new values of $\hat{p}_{ens}$ and $\hat{p}_{pro}$ need to be chosen – clearly, these should be greater than or equal to their lower bounds $\check{p}_{ens}$ and $\check{p}_{pro}$ respectively.

To test whether Equation (17) holds for the chosen $\hat{p}_{ens}$ and $\hat{p}_{pro}$, we need to compute the corresponding r_{ICS}, r_{car}, c_{ICS}, and c_{car}. We can compute r_{ICS} using Equation (6) and assuming that all processes are respectively released either at a $\hat{p}_{ens}$ or a $\hat{p}_{pro}$ rate[6].

$$r_{ICS} = 14\,\mathrm{ms} \times \left(\frac{0.025\,\mathrm{ms}}{7\,\mathrm{ms}} + 68 \cdot \frac{2 \times 0.025\,\mathrm{ms}}{14\,\mathrm{ms}} \right) + 0.025\,\mathrm{ms} + 68 \cdot (2 \times 0.025\,\mathrm{ms}) \approx 7\,\mathrm{ms}. \tag{18}$$

Similarly, we can compute r_{car} using Equation (7). In the car, there are only one ensemble process, one process to update the speed with the new one assigned by the ICS, and a knowledge propagation process. Again, we assume the ensemble process triggers the knowledge exchange process at cars. Assuming again $e_i = 25\,\mu\mathrm{s}$, we obtain:

$$r_{car} = 14\,\mathrm{ms} \times \left(\frac{0.025\,\mathrm{ms}}{7\,\mathrm{ms}} + \cdot \frac{2 \times 0.025\,\mathrm{ms}}{14\,\mathrm{ms}} \right) + (0.025\,\mathrm{ms} + 2 \times 0.025\,\mathrm{ms} = 0.18\,\mathrm{ms}.$$

Now we need compute $\hat{c}_E$, i.e., the transmission time on the Ethernet link taking contention into account, using Equation (11):

$$\hat{c}_E = \hat{p}_{pro} \times 68 \times \frac{1.168\,\mu\mathrm{s}}{\hat{p}_{pro}} + 68 \times 1.168\,\mu\mathrm{s} = 158.85\,\mu\mathrm{s},$$

where c_i and z_i have been replaced by c_E and $\hat{p}_{pro}$ respectively. Further, b_i is zero according to Equation (9), since we consider the lowest priority message for $\hat{c}_E$, i.e., the one suffering the most contention by other messages. The communication delay from and to the ICS, can be computed using Equation (12) and Equation (13):

$$c_{ICS} = 18.02\,\mu\mathrm{s}, c_{car} = 333.38\,\mu\mathrm{s}.$$

Finally, from Equation (2), we have that:

$$D_{max} = 2 \times 7\,\mathrm{ms} + 2 \times 14\,\mathrm{ms} + 0.3334\,\mathrm{ms} + 0.0181\,\mathrm{ms} + 7\,\mathrm{ms} + 0.18\,\mathrm{ms} \approx 50\,\mathrm{ms},$$

which is less than $t_{1\mathrm{m}} = 72ms$ – see Equation (14). That is, our ICS is able to meet all deadlines in the worst case.

[6]Note that processes with different rates are also possible; however, this is not meaningful in the context of our case study, where each process stands for an approaching car at the intersection.

7. Robustness under Unreliable Communication

In general, if many consecutive packets are lost on the communication channel, the system will experience malfunction putting safety into risk. In this section, we will determine the number of consecutive packets that can be lost at maximum without compromising safety. In other words, we quantify the system's *robustness* under unreliable communication.

As discussed above, Equation (2) states the worst-case delay incurred by our DEECo-based ICS in case of a fully reliable communication. This is obtained considering that data at a car can be updated immediately after knowledge has been propagated – for the reason that processes updating and propagating data are not synchronized with each other. As a result, this data will be sent the next time a knowledge propagation is performed, i.e., $\hat{p}_{pro}$ time later. If now this packet is lost on the communication channel, data will incur an additional delay equal to $\hat{p}_{pro}$. Further, if k_{car} denotes the number of consecutive packets that are lost, then data from the car to the ICS incurs the following delay:

$$\hat{p}_{pro} + k_{car} \times \hat{p}_{pro} + c_{car}, \tag{19}$$

where again c_{car} denotes the delay on the communication channel from the car to the ICS.

In a similar manner, if k_{ICS} denotes the number of consecutive packets that are lost from the ICS to the car, then data from the ICS incurs following delay to reach the car:

$$\hat{p}_{pro} + k_{ICS} \times \hat{p}_{pro} + c_{ICS}, \tag{20}$$

where, as discussed above, c_{ICS} is delay on the communication channel from the ICS to the car.

Let us denote by $k = k_{car} + k_{ICS}$ the total number of packets lost between the ICS and the car. We can now combine Equation (19) and Equation (20) to determine the closed-loop delay of the system in case of unreliable communication:

$$\hat{D}_{max} = (2 + k) \times \hat{p}_{pro} + 2 \times \hat{p}_{ens} + c_{ICS} + c_{car} + r_{ICS} + r_{car}, \tag{21}$$

where again r_{ICS} and r_{car} denote the maximum delay to finish computation at the ICS and the car respectively. Note that Equation (21) reduces to Equation (2) for $k = 0$, i.e., when no packets are lost on the communication channel.

Using the values of $\hat{p}_{pro}$, $\hat{p}_{ens}$, c_{car}, c_{ICS}, r_{car}, and r_{ICS} computed in the previous section, we can now determine the maximum k that can be tolerated without affecting the system's functionality and safety. That is the maximum k that makes $\hat{D}_{max}$ be at most equal to $t_{1\mathrm{m}}$ as per Equation (14). We denote this maximum k by k_{max}:

$$k_{max} = \left\lfloor \frac{t_{1\mathrm{m}} - D_{max}}{\hat{p}_{pro}} \right\rfloor = 3. \tag{22}$$

Equation (22) indicates that the sum of k_{car} and k_{ICS}, each of which represents the number of consecutive packets being lost in one or the other direction, cannot be more than 3 for the system to operate correctly.

Safety Mechanisms. Note that we can use the previous results to implement safety mechanisms at the ICS and at cars. In particular, whenever communication is lost for longer than $t_{1\mathrm{m}}$ time between the ICS and any car in the system or vice versa, both the ICS and the car switch to manual mode, i.e., the ICS starts working as standard traffic lights.

Priority levels	7
Message length	1024 bits
Packet send interval	7 ms ($\hat{p}_{pro}$ from the analysis)
ICS response delay	$\hat{p}_{pro} + \hat{p}_{ens} + r_{ICS} = 28$ ms
A car response delay	$\hat{p}_{pro} + \hat{p}_{ens} + r_{car} = 21.18$ ms
Bandwidth (Car to AP)	100 Mbps
Bandwidth (ICS to AP)	1 Gbps

Table 2: Simulation parameters

This can be implemented by DEECo processes that run at the different cars and at the ICS and trigger the manual mode in a decentralized manner. In other words, these processes behave as *watchdog timers* at the different nodes. They force a switch to manual mode at the corresponding node, if no packets have been received for longer than $t_{1\mathrm{m}}$ time. Note that here, for ease of exposition, we neglect the time which is necessary to process data at cars, i.e., $\hat{p}_{ens} + r_{car}$, and at the ICS, i.e., $\hat{p}_{ens} + r_{ICS}$. Whereas there is only one such watchdog processes at a car, there are multiple ones at the ICS; one for each car in the system.

It should be noticed that the ICS does not need to notify cars whenever it switches to manual mode; it suffices if it stops assigning speeds to them and cars themselves will automatically switch to manual mode. In the same way, if a car first switches to manual mode, the ICS will detect this on its own without need for notification from the car.

In the worst case, since we do not know which packets may be lost, we may have up to $3 \times t_{1\mathrm{m}}$ delay for the whole system, i.e., all cars and the ICS, to switch to manual mode in a decentralized manner. This results from considering the following conditions:

i) A packet from ICS is sent to arrive exactly $t_{1\mathrm{m}}$ time after its last packet at a given car.

ii) This packet from the ICS is lost at the communication channel such that the car (locally) switches to manual mode.

iii) All packets from the car to the ICS also get lost such that the ICS realizes that the car is in manual mode – and triggers itself a switch to manual mode – not until $t_{1\mathrm{m}}$ time later.

iv) All packets from the ICS to the remaining cars get lost such that, in the worst case, all other cars switch to manual mode $t_{1\mathrm{m}}$ time after the ICS.

As discussed above, this delay corresponds to 3 m in the trajectory of a car in our case study. Hence the ICS has to assign speeds to cars – in the automatic mode – such that there is sufficient distance between them taking vehicle dynamics into account (e.g., if a car suddenly breaks, it will not stop immediately due to its inertia, etc.).

Finally, only the ICS can decide to go back to the automatic mode whenever communication to all cars has normalized. To this end, the ICS needs to notify or start assigning speeds to all cars in the system. Note that the delay for switching to the automatic mode is given by $t_{1\mathrm{m}}$ since we assume a normal communication.

8. Experimental Evaluation

In this section, we validate the analysis presented in Section 6 by means of simulation. To this end, we created an OMNet++ simulation [56] using *INET* hardware models.

We have set up an OMNet++ simulation by manually implementing DEECo components as OMNet++ modules. In particular, we have implemented an OMNet++ module for each vehicle at the intersection and for the ICS. While the ICS is stationary, vehicles and their corresponding modules in OMNet++ move with given speeds. The modules generate network traffic that emulates the communication of vehicles entering and exiting the ICS's region of influence. This reflects the knowledge/data propagation for our DEECo-based ICS, from which we collect end-to-end communication latencies for a large set of simulated packet transmissions.

Our network topology consists of one ICS host connected by a full-duplex switch to three AP – see Figure 6. Vehicles connect dynamically to the AP adjusting message priorities as they get closer to the intersection. The communication from the switch to the ICS host is performed under message prioritization according to the Ethernet 802.1Q standard. Our simulation scenario spans different numbers of vehicles (20, 50 and 70 correspondingly) exchanging packets with the ICS. Table2 summarizes the most important simulation parameters considered in our evaluation.

Figure 7 and Table 3 show the results of our simulation with respect to closed-loop reaction time – i.e., the Car-ICS-Car delay – and for an increasing number of consecutive packet losses at the communication channel. In the case that no packets are lost, this figure shows that our $D_{max} = 50\,\text{ms}$ – computed at the end of Section 6 – is safe. That is, in this case, all delay values in the system are always less than 50 ms even for 70 cars, i.e., two more cars than what it is considered and allowed by the analysis presented in the above sections.

Evaluation under Unreliable Communication. Now, we discuss our simulation results for a varying number of consecutive packet losses either from the car to the ICS or from the ICS to the car. As it can be observed in Figure 7, the system operates properly – i.e., the Car-ICS-Car delay is below below $t_{1\text{m}} = 72\,\text{ms}$ – for up to 3 consecutive packet losses, which validates our analysis in Section 7. Clearly, the more packets are lost, the higher the Car-ICS-Car delay is; however, this is always less than the computed threshold $t_{1\text{m}}$ and, hence, the system can remain in automatic mode.

For the case of 4 packets lost, also depicted in Figure 7, the Car-ICS-Car delay starts exceeding the threshold $t_{1\text{m}} = 72\,\text{ms}$ – even when considering only 20 cars at the intersection. As a result, the system cannot tolerate more than 3 consecutive packet losses without switching to manual mode. This again is in accordance with the computed upper bound on packet losses given in Equation (22).

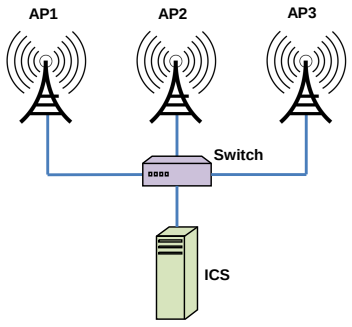

Figure 6: Simulated network consisting of three access points (APs) and a switch

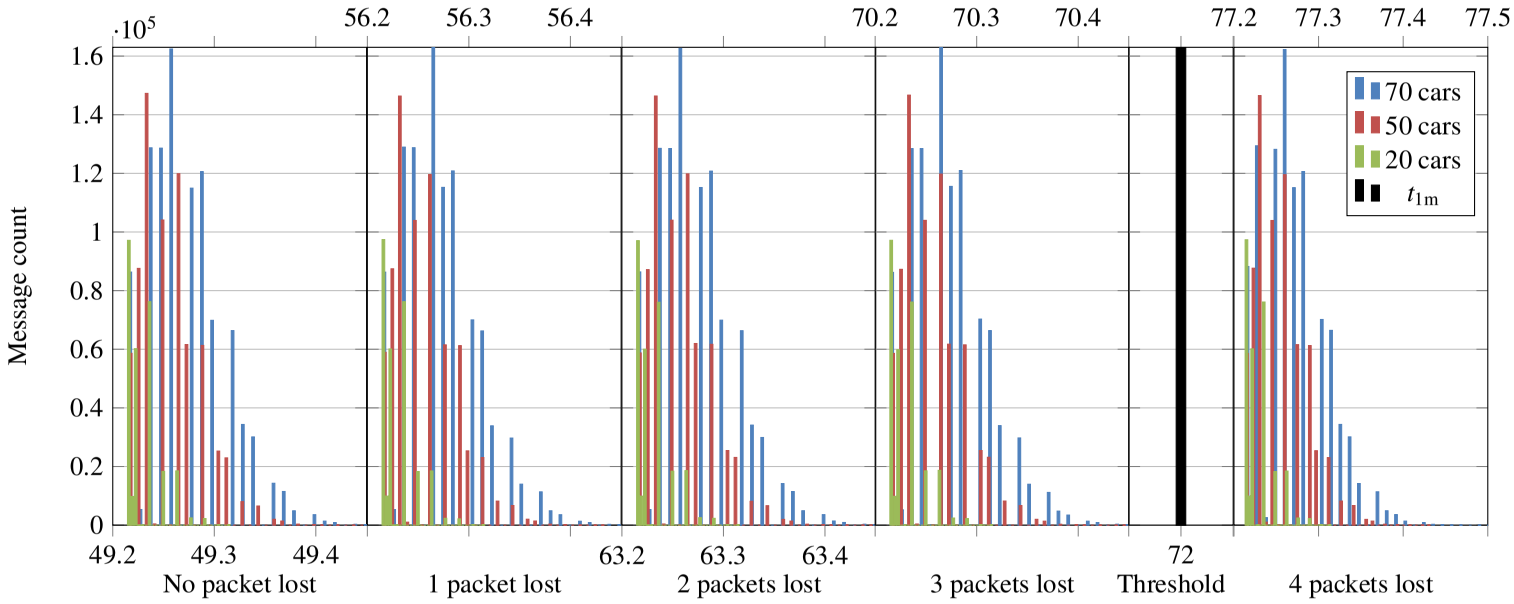

Figure 7: Car-ICS-Car closed-loop reaction times in milliseconds

No packet lost	20 vehicles	50 vehicles	70 vehicles
Mean	49.2245	49.2507	49.2694
Std. Dev.	0,0157	0.00298	0.03723
Median	49.2182	49.2449	49.2582
Max	49.3117	49.4453	49.5121
1 packet lost			
Mean	56.2245	56.2507	56.2692
Std. Dev.	0,0155	0.00299	0.0371
Median	56.2182	56.2449	56.2582
Max	56.3117	56.4319	56.4987
2 packets lost			
Mean	63.2246	63.2508	63.2694
Std. Dev.	0,0156	0.00298	0.0372
Median	63.2182	63.2449	63.2582
Max	63.3117	63.4453	63.5121
3 packets lost			
Mean	70.2246	70.2509	70.2693
Std. Dev.	0,0156	0.00298	0.0371
Median	70.2182	70.2449	70.2582
Max	70.3117	70.4453	70.4987
4 packets lost			
Mean	77.2245	77.2507	77.2694
Std. Dev.	0,0156	0.00298	0.0372
Median	77.2182	77.2449	77.2582
Max	77.3117	77.4319	77.5388

Table 3: Reaction time statistics (values given in milliseconds)

Realism of the Evaluation. The presented results are based on a simulation and, thus, they may differ in reality. In particular, we have made a number of assumptions which may not hold and, hence, have an impact on our evaluation. In the following, we discuss this in more detail.

- The computed $t_{1\mathrm{m}}$ may not hold. This is based on the assumption that cars/vehicles can have speeds of up to 50 Km/h – see (14). However, in reality, it may happen that one or more cars exceed this speed limit by some amount. A solution to this is to consider a safety margin and, for example, compute a new $t_{1\mathrm{m}}$ for 60 Km/h instead. However, it now may happen that the ICS cannot meet this deadline anymore. To overcome this problem, the number of cars at the intersection can be restricted to a safe value. If more cars than safe enter the ICS's region of influence, it will switch to manual mode. Clearly, this higher speed limit can also be exceeded. In this case, the ICS can directly switch to manual mode.

- The computed maximum number of cars at the intersection n may also not hold. This is based on assumptions on the minimum length of cars and on the maximum possible distance between any two cars at the intersection – see (15). If these assumptions do not hold in practice, the maximum number of cars at the intersection may potentially increase. This has impact on the WCRT of the ICS r_{ICS} and on the worst-case communication delay from a car to the ICS c_{car}. As a result, the ICS may probably not be able to meet deadlines

any longer and, hence, it will have to switch to manual mode to guarantee safety, if more cars than the maximum expected enter its region of influence.

- The WCET of processes at the cars and at the ICS may be greater than the assumed $e_i = 50\,\mu$s. This will have direct impact on the WCRT at the car r_{car} and at the ICS r_{ICS}. As a consequence, the ICS may not be able to meet deadlines anymore and, again, it will have to switch to manual mode, if a given number of cars is exceeded at the intersection.

- The bandwidths assumed for the different segments (either from the car to the AP or from the AP to the ICS) are less than those assumed in Table 2. This leads to increased communication delays in both directions from the car to the ICS and vice versa. The ICS may stop being able to meet deadlines and, thus, it will have to restrict the number of cars at the intersection in the automatic mode.

From the above discussion, it should be clear that we can account for discrepancies between our simulated and a real-life ICS by taking a conservative estimate on the maximum number of cars that the ICS can simultaneously handle. If, in practice, this number is exceed, the ICS will switch to manual mode preserving safety at the cost of restricting service.

9. Concluding Remarks

In this paper, we have presented DEECo as a special-purpose, component-based, design and development framework for open-ended CPS. DEECo specifically targets at dynamic distributed systems and, thus, provides systematic software engineering mechanisms to describe and analyze such complex application scenarios. These mechanisms mainly consist in modeling transitory interactions between one or more components in the system.

We extended DEECo's design flow by a technique to estimate worst-case, closed-loop, response times between DEECo components. This effectively allows guaranteeing real-time requirements from high-level DEECo-based designs provided that the underlying platform supports real-time, e.g., real-time OS, priority-based communication protocols, etc. Clearly, if the underlying technologies are nondeterministic, then it is not possible to provide any timing guarantees and, as a consequence, no safety-critical applications can be implemented on their basis.

We illustrated our proposed technique based on an intelligent crossroad scenario. Towards this, we derived the worst-case delay D_{max} of a DEECo-based system – see Equation (2). This analysis is general enough and can be used for other applications. Note that the term $2 \times \hat{p}_{pro} + 2 \times \hat{p}_{ens}$ is the overhead by DEECo, whereas $c_{ICS} + c_{car}$ and $r_{ICS} + r_{car}$ stand for the communication and the computation overhead respectively. DEECo's overhead is configurable by properly choosing $\hat{p}_{pro}$ and $\hat{p}_{ens}$ which again need to be in accordance with the application requirements. The communication and computation overhead will depend on the used technologies such as communication protocols, scheduling algorithms, etc.

Based on our analysis, we evaluated the robustness of a DEECo-based design against packet losses at the communication channel. Towards this, we analytically obtained a upper bound on the number of packets that can be lost without affecting the system's safety. We further validated this bound by means of extensive experiments based on an OMNet++ simulation. In addition, we proposed and discussed safety mechanisms that can be integrated into a DEECo design in order to adapt to unpredicted communication loss between components.

We envision integrating the proposed technique into the existing ensemble development life cycle [57], which provides a systematic approach (i.e., methodology) towards engineering open-ended CPS. The presented work fits into the modeling part of that cycle, which is followed by verification performed on the basis of simulation techniques – similar to the procedure shown in this paper. An important aspect is also the requirements engineering part, which should besides functional properties also account for extra-functional, in particular, real-time aspects.

Overall, the technique presented in this paper allows reasoning about real-time requirements at the component level and constitutes a necessary step towards holistic software engineering methods for modern cyber-physical systems.

Acknowledgments

This work was partially supported by Charles University institutional funding SVV-2014-260100. The research leading to these results has received funding from the European Union Seventh Framework Programme FP7-PEOPLE-2010-ITN under grant agreement n°264840.

Appendix A. DEECo-based ICS implementation

In the following, we discuss the implementation of our ICS case study. This is based on the CDEECo library (C++ version of DEECo). The sources of the case study draft and network traffic simulation (in OMNet++) are placed on the GitHub.[7] We refer to the README.md file in the root directory of the repository for more details.

Appendix A.1. DEECo to C++ mapping

We discuss how DEECo concepts – such as component, ensemble, knowledge and process – are mapped to C++. Thereby, type safety and real-time operation need to be guaranteed.

Components. A component is mapped to a C++ class that inherits from *CDEECO::Component* template parametrized by *knowledge type*. The base class is responsible for knowledge access and distribution, including the fragmentation of knowledge into packets. However, the base class does not perform any broadcasting of packets, which is done by another class inheriting from *CDEECO::Broadcaster*. A component class specifies the type of component and identification number.

Some extra methods and fields (not related to DEECo) can be added into the component class. This is the ideal place to store component-related external data.

```
1  class Component: public CDEECO::Component<Knowledge> {
2  public:
3      static const CDEECO::Type type = 1 ;
4  private:
5      // Process handles here
6      // User data here
7  Component (CDEECO::Broadcaster &broadcaster , const CDEECO::Id id) :
8      CDEECO::Component<Knowledge>( id , type , broadcaster) {
9          // Knowledge initialization here
10     }
11 };
```

Listing 5: Component example

[7] https://github.com/d3scomp/ICS-CDEECo

Knowledge type. Knowledge is a plain C++ structure. This approach has some limitations compared to wrapping knowledge parts into a richer class structure, but it simplifies the user-defined code, gives guarantees on complexity and speeds up execution.

```
struct Knowledge: CDEECO::Knowledge {
    uint32_t foo;
    uint32_t bar;
};
```

Listing 6: Knowledge example

```
class ExampleProcess:
public CDEECO::PeriodicTask<ExampleKnowledge, int> {
public:
    ExampleProcess (auto &component):
        PeriodicTask(2400 , component , component.knowledge.value ) {
        // User initialization here
    }
private:
    // User data here
    // User task code, executed every 2400ms
    int run (const ExampleKnowledge in) {
        return in.value + 42;
    }
};
```

Listing 7: Process example

```
// Typedef base ensemble type
typedef CDEECO::Ensemble<Coord::Knowledge, Coord::Knowledge::Id, Member::Knowledge,
    Member::Knowledge::Id> EnsembleType ;
// Id exchange ensemble
class Ensemble: EnsembleType {
    public:
    // Define ensemble period
    static const auto PERIOD MS = 42;
    Ensemble (CDEECO::Component<Coord::Knowledge> &coordinator, auto &library): EnsembleType(&coordinator,
        &coordinator.knowledge.id, &library, PERIOD MS) {}
    Ensemble (CDEECO::Component<Member::Knowledge> &member, auto &library): EnsembleType(&member ,
        &member.knowledge.id , &library, PERIOD MS) {}
    protected:
        // Membership condition
        bool isMember (const CDEECO::Id coordId, const Coord::Knowledge coordKnowledge, const CDEECO::Id memberId,
            const Member::Knowledge memberKnowledge) {
            // When coordinator has lower id the membership success
            return coordId < memberId ;
        }

        // Map member id to coordinator
        Coord::Knowledge::Id memberToCoordMap(const Coord::Knowledge coord, const CDEECO::Id memberId,
            const Member::Knowledge memberKnowledge) {
            return memberId ;
        }

        // Map coordinatorId to member
        Member::Knowledge::Id coordToMemberMap(const Member::Knowledge member, const CDEECO::Id coordId,
            const Coord::Knowledge coordKnowledge) {
            return coordId ;
        }
}
```

Listing 8: Ensemble example

Process and ensemble. A process is defined by a class inheriting from descendants of the class template *CDEECO::Task* – either *CDEECO::PeriodicTask* or *CDEECO::TriggeredTask* class templates. Inputs and outputs of the process are defined by template arguments, thus, type safety

needs to be ensured in the user code. Similar to the component class, the user-defined process class can hold unrelated data not managed by CDEECo.

Ensembles are handled in a very similar way as processes. That is, an ensemble is just a process which is using knowledge data of two components. Technically, an ensemble inherits from *CDEECO::Ensemble* class template. The only major difference is that an ensemble condition together with a knowledge mapping method needs to be defined. The ensemble definition further needs to establish the ensemble type based on the knowledge of the components involved.

Parameter passing. In order to provide type safety, templates are used for processes and ensemble mapping functions. Component and ensemble classes have their input and output knowledge types as their template arguments. Thus the user-defined functions can receive their inputs and return their outputs directly as members of the *knowledge* structure. There is no need to cast from generic types.

```
1 Mode SampleProcess::run(const Knowledge in) {
2     if(in.foo == in.bar) {
3         return Mode::Manual;
4     else
5         return Mode::Automatic;
6 }
```

Listing 9: Process parameter access

Appendix A.2. Runtime environment

CDEECo provides a runtime environment for DEECo applications. As mentioned before, it is internally using FreeRTOS for scheduling processes and ensembles. In fact, C++ wrappers are implemented, which allow for a straightforward portability to another real-time OS. FreeRTOS has the advantage of supporting many relevant microcontrollers/boards used in embedded systems.

Another property critical in real-time systems is memory management. In general, it is difficult to provide a time bound for allocating and deallocating memory from the heap. As a result, in CDEECo, all data structures are either fully static or allocated on the heap at the initialization phase.

The whole CDEECo runtime environment relies on C++ class templates. These are generally used to provide a clean and typed API to the designer. Moreover, using C++ templates allows for a more predictable running time since they are generally resolved at compile time.

Appendix A.3. Deployment

Our implementation of the ICS case study can be deployed on the STM32F4 board[8] extended with the IEEE802.15.4 interface. The application together with some board specific source files is statically linked with the CDEECo library. Particularly, the library expects the application to define the symbol *cdeecoSetup*. The resulting binary representing either the ICS or a vehicle node is then flashed onto the board.

Compiling and linking sources is straightforward. Since CDEECo requires some of the recent C++14 features, it is necessary to use recent compiler and libc versions. For development, a custom toolchain compiled from sources was used. The toolchain was created on the Gentoo[9] system using the *crossdev* utility.

[8] `http://st.com/stm32f4`

[9] `http://gentoo.org`

```
$ crossdev --target armv7m-hardfloat-eabi --ex-gcc --ex-gdb
```

These software versions have been used:

- binutils: 2.24-r3 with C++ support
- gcc: 4.9.0 with C++ support
- gdb: 7.7.1 with XML support
- newlib: 2.1.0
- openocd: 0.9.0

Build instructions. It might be necessary to adapt makefiles in case that the toolchain and *openocd* are located in a different place. Once the tools are located, compilation requires just one command to be executed from the project's root directory:

```
$ make
```

Note that the makefiles currently do not support a parallel build. If no errors are encountered, two application binaries will be created. The binaries are automatically dumped as *hex* files that can be flashed onto the development board using *openocd*. The ICS and vehicle applications can be flashed with following commands executed from repository's root directory.

```
$ make flash-ics
$ make flash-vehicle
```

Appendix A.4. Modeling with DEECo

Next, we show an example of a DEECo model for the ICS case study.

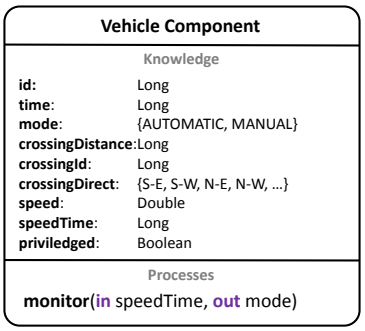

Figure A.1: Vehicle component specification

DEECo components. Figure A.1 specifies the vehicle component that is characterized by the set of attributes (together with their types) listed in the figure and the following process:

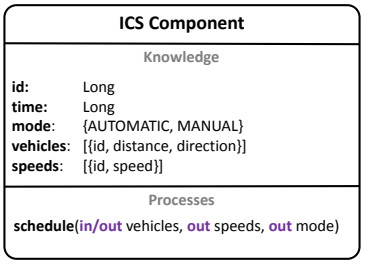

Figure A.2: ICS component specification

- *monitor(in speedTime, out mode)*
  The process is responsible for monitoring whether input data is obsolete or not (i.e., the time of the last speed update must be less than a *threshold* given by our real-time analysis in Section 6). If so, i.e., if the value from *ICS* is obsolete, then *mode* is set to the MANUAL. As input parameter, it takes the *ICS* identifier, whereas it returns the vehicle's *mode*.

Figure A.2 specifies the ICS component that is characterized by the set of attributes (together with their types) listed in the figure and the following process:

- *schedule(in/out vehicles, out arrivals, out mode)*
  This process is responsible for computing and monitoring the speeds of approaching vehicles/cars depending on the current traffic situation. Towards this, given cars' current speeds and directions, it computes the time at which they reach the intersection. Taking cars' lengths and widths into account, it adjusts their speeds to avoid conflicts. Which car is allowed to cross first depends on the order in which they arrive at ICS's region of influence and on whether they are privileged or not. If the ICS detects that a car or vehicle does not maintain the assigned speed, it changes the value of *mode* to MANUAL, i.e., it starts working as standard traffic lights. The same happens, if communication is lost to one or more cars. As input parameters, this process takes *vehicles*, i.e., a collection of the most recent states of cars/vehicles at the crossing. As output parameters, it returns *vehicles*, where the *speed* attribute of each vehicle is updated, and *mode*.

Figure A.3 and Figure A.4 show the specification of ensembles, i.e., interactions or relations, between the ICS and Vehicle components. The attributes of the *coordinator* in these ensembles match the attributes of the ICS component, while the *member's* attributes match the vehicle component. Note that, in the end, this is the same interaction/relation but specified from the *perspective* of the member – see Figure A.3 – and of the coordinator – see Figure A.4.

[1] K. Beetz, W. Böhm, Challenges in engineering for software-intensive embedded systems, in: Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology, Springer, 2012.
[2] AUTOSAR: Layered software architecture, `http://autosar.org/download/R4.0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf`.
[3] W. Reisig, Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies, Springer, 2013.
[4] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, DEECo: An ensemble-based component system, in: Proceedings of the International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE), 2013.
[5] R. Hennicker, A. Klarl, Foundations for ensemble modeling – the Helena approach, in: Specification, Algebra, and Software, Springer, 2014.
[6] A. Masrur, M. Kit, T. Bures, W. Hardt, Towards component-based design of safety-critical cyber-physical applications, in: Proceedings of the Euromicro Conference on Digital Systems Design (DSD), 2014.

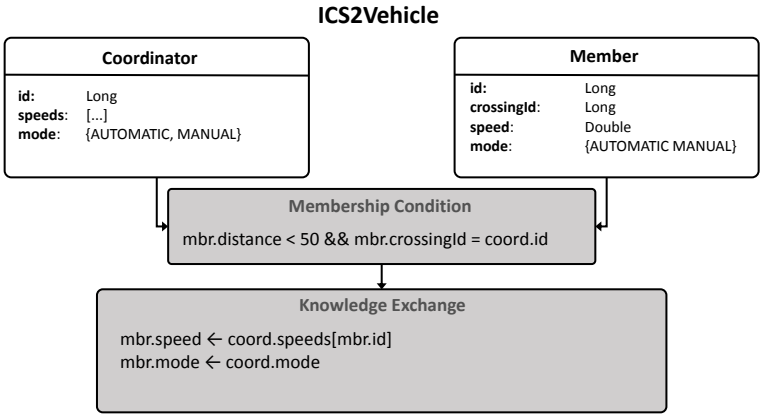

Figure A.3: ICS2VehicleEnsemble specification

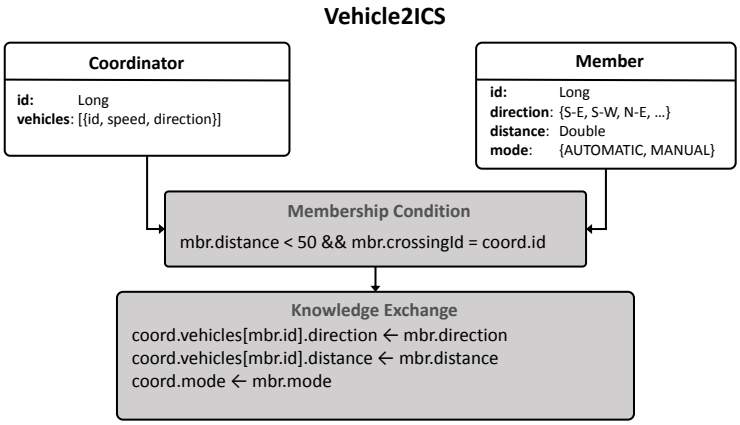

Figure A.4: Vehicle2ICSEnsemble specification

[7] J. Keznikl, T. Bures, F. Plasil, I. Gerostathopoulos, P. Hnetynka, N. Hoch, Design of ensemble-based component systems by invariant refinement, in: Proceedings of the International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE), 2013.

[8] C++ implementation of the DEECo runtime environment, `https://github.com/d3scomp/CDEECo`.

[9] M. Kit, I. Gerostathopoulos, T. Bures, P. Hnetynka, F. Plasil, An architecture framework for experimentations with self-adaptive cyber-physical systems, in: Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015.

[10] S. Zeadally, R. Hunt, Y.-S. Chen, A. Irwin, A. Hassan, Vehicular ad hoc networks (VANETS): status, results, and challenges, Telecommunication Systems 50 (4).

[11] J. Feljan, L. Lednicki, J. Maras, A. Petricic, I. Crnkovic, Classification and survey of component models, Tech. rep., Project DICES (Feb. 2009).

[12] T. Pop, P. Hnetynka, P. Hosek, M. Malohlava, T. Bures, Comparison of component frameworks for real-time embedded systems, Knowledge and Information Systems 40 (1).

[13] jRESP – runtime environment for SCEL programs, `http://www.ascens-ist.eu/jresp/`.

[14] K. Klobedanz, C. Kuznik, A. Thuy, W. Mueller, Timing modeling and analysis for AUTOSAR-based software development - a case study, in: Proceedings of Conference on Design, Automation, and Test in Europe (DATE), 2010.

[15] J. E. Kim, O. Rogalla, S. Kramer, A. Hamann, Extracting, specifying and predicting software system properties in component based real-time embedded software development, in: Proceedings of the International Conference on Software Engineering (ICSE), 2009.
[16] R. Ommering, F. Linden, J. Kramer, J. Magee, The Koala component model for consumer electronics software, Computer 33 (3).
[17] H. Maaskant, A robust component model for consumer electronic products, in: Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices, Springer, 2005.
[18] C. Sunder, A. Zoitl, J. Christensen, H. Steininger, J. Rritsche, Considering IEC 61131-3 and IEC 61499 in the context of component frameworks, in: Proceedings of the IEEE International Conference on Industrial Informatics (INDIN), 2008.
[19] L. Lednicki, J. Carlson, K. Sandström, Model level worst-case execution time analysis for IEC 61499, in: Proceedings of the International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE), 2013.
[20] J.-P. Fassino, J.-B. Stefani, J. Lawall, G. Muller, Think: A software framework for component-based operating system kernels, in: Proceedings of the USENIX Annual Technical Conference, 2002.
[21] M. Anne, R. He, T. Jarboui, M. Lacoste, O. Lobry, G. Lorant, M. Louvel, J. Navas, V. Olive, J. Polakovic, M. Poulhies, J. Pulou, S. Seyvoz, J. Tous, T. Watteyne, Think: View-based support of non-functional properties in embedded systems, in: Proceedings of International Conference on Embedded Software and Systems (ICESS), 2009.
[22] The MIND project, `http://mind.ow2.org/`.
[23] The Fractal Component Model specification, `http://fractal.ow2.org/specification/`.
[24] M. Prochazka, R. Ward, P. Tuma, P. Hnetynka, J. Adamek, A component-oriented framework for spacecraft onboard software, in: Proceedings of Data Systems in Aerospace (DASIA), 2008.
[25] T. Bures, P. Hnetynka, F. Plasil, SOFA 2.0: Balancing advanced features in a hierarchical component model, in: Proceedings of International Conference on Software Engineering Research, Management and Applications (SERA), 2006.
[26] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, T. Sivaharan, A generic component model for building systems software, ACM Transactions on Computer Systems 26 (1).
[27] S. Hissam, J. Ivers, D. Plakosh, K. C. Wallnau, PIN component technology (v1.0) and its C interface, Tech. rep., CMU SEI (Apr. 2005).
[28] O. Sokolsky, A. Chernoguzov, Performance analysis of AADL models using real-time calculus, in: Foundations of Computer Software: Future Trends and Techniques for Development, Springer, 2010.
[29] L. Thiele, S. Chakraborty, M. Naedele, Real-time calculus for scheduling hard real-time systems, in: Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), 2000.
[30] OMG unified modeling language, version 2.5, `http://omg.org/spec/UML/2.5/`.
[31] A. Louati, K. Barkaoui, C. Jerad, Temporal properties verification of real-time systems using UML/MARTE/OCL-RT, in: Formalisms for Reuse and Systems Integration, Springer, 2015.
[32] OMG systems modeling language, version 1.3, `http://omg.org/spec/SysML/1.3/`.
[33] H. Espinoza, D. Cancila, B. Selic, S. Gerard, Challenges in combining SysML and MARTE for model-based design of embedded systems, in: Model Driven Architecture - Foundations and Applications, Springer, 2009.
[34] E. Andrade, P. Maciel, G. Callou, B. Nogueira, A methodology for mapping SysML activity diagram to time petri net for requirement validation of embedded real-time systems with energy constraints, in: Proceedings of the International Conference on Digital Society (ICDS), 2009.
[35] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. P. Black, P. O. Müller, C. Zeidler, T. Genssler, R. Born, A component model for field devices, in: Component Deployment, Springer, 2002.
[36] X. Ke, K. Sierszecki, C. Angelov, COMDES-II: A component-based framework for generative development of distributed real-time control systems, in: Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), 2007.
[37] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, K. Lundbäck, The Rubus component model for resource constrained real-time systems, in: Proceedings of the IEEE International Symposium on Industrial Embedded Systems (SIES), 2008.
[38] H. Hansson, M. Åkerholm, I. Crnkovic, M. Törngren, SaveCCM – a component model for safety-critical real-time systems, in: Proceedings of EUROMICRO Conference, 2004.
[39] T. Bures, J. Carlson, I. Crnkovic, S. Sentilles, A. Vulgarakis, ProCom – the progress component model reference manual, version 1.0, Tech. rep., Mälardalen University (Jun. 2008).
[40] MyCCM high integrity, `https://srcdev.lip6.fr/trac/research/flex-eware/wiki/MyCCM`.
[41] S. Becker, H. Koziolek, R. Reussner, The palladio component model for model-driven performance prediction, Systems and Software 82 (1).
[42] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, R. E. Schantz, Total quality of service provisioning in middleware and applications, Microprocessors and Microsystems 27 (9–10).
[43] A. Basu, M. Bozga, J. Sifakis, Modeling heterogeneous real-time components in BIP, in: Proceedings of the IEEE

International Conference on Software Engineering and Formal Methods (SEFM), 2006.
[44] C. Etzien, T. Gezgin, S. Froschle, S. Henkler, A. Rettberg, Contracts for evolving systems, in: Proceedings of the IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013.
[45] T. Gezgin, C. Etzien, Correct by prognosis: Methodology for a contract-based refinement of evolution models, in: Complex Systems Design & Management (CSD&M), 2014.
[46] T. Gezgin, S. Henkler, A. Rettberg, I. Stierand, Contract-based compositional scheduling analysis for evolving systems, in: Embedded Systems: Design, Analysis and Verification, Springer, 2013.
[47] T. Bures, I. Gerostathopoulos, P. Hnetynka, J. Keznikl, M. Kit, F. Plasil, Gossiping components for cyber-physical systems, in: Software Architecture, Springer, 2014.
[48] C. Liu, J. Layland, Scheduling algorithms for multiprogramming in hard real-time environments, Journal of the Association for Computing Machinery 20 (1).
[49] N. Audsley, A. Burns, M. Richardson, K. Tindell, A. Wellings, Applying new scheduling theory to static priority pre-emptive scheduling, Software Engineering Journal 8 (5).
[50] IEEE 802.11p standard: Wireless LAN MAC and PHY specifications amendment 6: Wireless access in vehicular environments, `http://standards.ieee.org/findstds/standard/802.11p-2010.html`.
[51] IEEE 802.1Q standard: LANs and WANs – MAC bridges and virtual bridged LANs, `http://standards.ieee.org/findstds/standard/802.1Q-2011.html`.
[52] N. Shah, F. Bastani, S. Kumar, I.-L. Yen, Real-time car-to-car communication protocol for intersecting roads, in: Proceedings of the International Conference on ITS Telecommunications (ITST), 2008.
[53] N. Shah, S. Kumar, F. Bastani, I.-L. Yen, Optimization models for assessing the peak capacity utilization of intelligent transportation systems, European Journal of Operational Research 216 (1).
[54] S.-Y. Pyun, H. Widiarti, Y.-J. Kwon, D.-H. Cho, J.-W. Son, TDMA-based channel access scheme for V2I communication system using smart antenna, in: Proceedings of the IEEE Conference on Vehicular Networking (VNC), 2010.
[55] S.-Y. Pyun, H. Widiarti, Y.-J. Kwon, J.-W. Son, D.-H. Cho, Group-based channel access scheme for a V2I communication system using smart antenna, IEEE Communications Letters 15 (8).
[56] A. Varga, R. Hornig, An overview of the OMNeT++ simulation environment, in: Proceedings of the International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTOOLS), 2008.
[57] T. Bures, R. D. Nicola, I. Gerostathopoulos, N. Hoch, M. Kit, N. Koch, G. V. Monreale, U. Montanari, R. Pugliese, N. Serbedzija, M. Wirsing, F. Zambonelli, A life cycle for the development of autonomic systems: The e-mobility showcase, in: Proceedings of the Workshop on Challenges for Achieving Self-Awareness in Autonomic Systems (AWARENESS), 2013.