Model Problem and Testbed for Experiments with Adaptation in Smart Cyber-Physical Systems

Vladimir Matena[1], Tomas Bures[1,3], Ilias Gerostathopoulos[1,2], Petr Hnetynka[1]

[1] Charles University in Prague
Faculty of Mathematics and Physics
Prague, Czech Republic

[2] Fakultät für Informatik
Technische Universität München
Munich, Germany

[3] Institute of Computer Science
The Czech Academy of Sciences
Prague, Czech Republic

{matena, bures, iliasg, hnetynka}@d3s.mff.cuni.cz

ABSTRACT

In this artifact, we partially address the problem of development of smart Cyber-Physical Systems (sCPS) by providing a concrete model problem and testbed for experimenting with, comparing, and developing new adaptation techniques and algorithms pertinent to sCPS. In particular, our model problem features autonomous robots cooperating opportunistically in a highly dynamic environment with multiple sources of uncertainty and runtime failures. Our testbed provides ROS-based Stage simulation of the model problem reified in a swarm of Turtlebot robots. The testbed ties this to timing-, bandwidth- and mobility-aware simulation of the robot communication (based on OMNeT++). To enable fast prototyping, the testbed abstracts robots as autonomous components (implemented in Java) and allows describing robot communication via dynamic collaboration groups (ensembles). It also points to specific places in the simulation code where adaptation logic can be plugged in.

CCS Concepts

• **Computer systems organization~Embedded and cyber-physical systems;** • *Computing methodologies~Simulation environments;* • *Software and its engineering~Software development techniques;*

Keywords

Smart cyber physical systems, self-adaptation, model problem, testbed

1. INTRODUCTION

Smart Cyber-Physical Systems (sCPS) are distributed and decentralized systems that closely cooperate with their physical environment by sensing and actuating [8]. A characteristic feature of sCPS is that they exhibit a high level of "intelligence" in terms of opportunistic cooperation, dynamic self-organization, self-healing and self-adaptation [5]. As such, sCPS are regarded as vital for building applications for smart mobility, smart energy grids, ambient assisted living, smart cities, etc.

SEAMS'16, May 16-17, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4187-5/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897053.2897065

Software engineering of sCPS is largely an open challenge, as sCPS combine autonomous decentralized cooperative behavior, with concerns of real-time, limited communication, dependability, etc. The lack of software engineering support also applies to self-adaptation [6], which is a central feature of sCPS, crucial for coping with the uncertain environments in which sCPS operate.

While there is a large body of knowledge for experimenting with adaptation in the context of enterprise services and other traditional software systems, there is rather a vacuum in terms of knowledge and especially tools for experimenting with adaptation in sCPS. This is in our view because sCPS combine multiple relatively distinct disciplines (real-time, control, networking, agents, learning, data-analysis, etc.) [4]. This consequently requires engineering approaches and tools for sCPS to build synergies between the disciplines and support the mutual interplay of the concerns.

In this artifact paper, we partially address the problem of development of self-adaptive sCPS by providing a model problem and testbed for experimenting with, comparing, and developing new adaptation solutions pertinent to sCPS.

In particular, our model problem and testbed provide challenges in coordination of autonomous robots with the interplay of concerns of (a) realistic communication (i.e., communication limited by bandwidth and subject to latencies), (b) real-time control, and (c) decentralized operation.

To enable fast prototyping, the testbed abstracts robots as autonomous components (implemented in Java) and allows describing robot communication via dynamic collaboration groups. It also points to specific places in the simulation code where adaptation logic can be plugged in and provides metrics for evaluating the plugged-in adaptation. Thus, together, the model problem and the testbed provide a concrete ready-to-use benchmark for experiments in the relatively new field of sCPS.

The paper is organized as follows. Section 2 describes the model problem in detail. Section 3 presents the testbed from both the user-perspective and also implementation point of view. Section 4 describes a sample adaptation we have used for evaluating the testbed and further discusses lessons learned and limitation. Section 5 briefly details the structure of the provided artifact (detailed instructions are packaged together with the software artifact). Section 6 discusses related work while Section 7 concludes the paper by summarizing the contributions.

2. MODEL PROBLEM

The reference problem provided by our testbed is the "Autonomous Cleaning Robots Coordination" (ACRC) problem. In ACRC, a number of cleaning robots is deployed in in-door space consisting of corridors and multiple office rooms (see Figure 1).

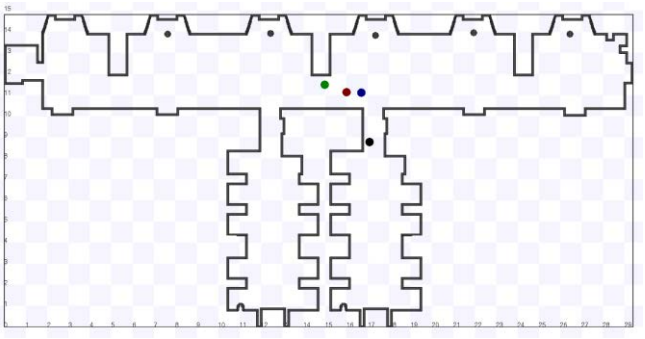

Figure 1. A visualization of the model problem.

Every robot is equipped with a 360 degrees camera which provides depth information. The robots uses the camera to observe obstacles (other robots, walls, etc.) and for navigation, by means of Adaptive Monte-Carlo Localization (AMCL). Robots are equipped with a map of the place that they are supposed to clean. This map is used in the AMCL-based navigation, which works by comparing a depth scan with the map.

Robots are capable of limited communication using an IEEE 802.15.4 transceiver (with approx. 10 meter direct visibility range), which allows building mobile ad-hoc networks. This means that robots can exchange data only when they are close to one another. Robots can extend the communication range by acting as proxies that rebroadcast messages further. Generally, however, no global communication can be assumed as situations when no proxy is close enough or too much interference exists are rather often.

The basic software of the robots is formed by Robot Operating System[1] (ROS), which is the de-facto standard set of libraries and services for building open-source robotic platforms.

2.1 Operation and Adaptation Challenges

Each robot is initially given its own set of places it is supposed to visit and clean. In the naïve solution, which can be considered as the baseline, robots act completely independently of one another (i.e., they do not communicate nor coordinate) and visit places on their list in the given order.

Due to the complexity of the environment and the deficiencies in the ROS stack (which we consider as a black-box component that is given and one has to live with), the naïve solution gives rise to multiple problems:

- A robot has only an approximation of its position and orientation. Often, especially when other robots are present nearby, the AMCL localization fails as the depth scans (which include other robots) cannot be matched with the known map. As a result, the robot navigation becomes very imprecise and sometimes, when in dense traffic, fails completely and the robot stops.
- The navigation module in a robot sometimes fails to find a route to the destination because other robots moving by obstruct it. As the result the robot stops.
- Due to physical space constraints, robots often get to a deadlock situation – e.g., when one robot wants to enter the office and another wants to exit it. The result is again that the robots stop to avoid collision. (Note that this is a different situation to the previous point, where the failure to find a way is only transient. In this case, however, it persists until the deadlock is explicitly solved.)

Generally, each of these problems can be solved by pointing the robot to the right direction. However, it practically turns out to be quite difficult to (1) distinguish the cause of the problem, and (2) to know where to navigate the robot to recover it from the failed state.

Though these problems could be targeted by modifying ROS, our experience with extending and customizing ROS shows that a more practically viable solution is to regard ROS as a black-box and build an adaptation layer over it. As such, the robotic scenario constitutes an excellent case for adaptation. (Of course, this is by no way a criticism of ROS, which itself is the most comprehensive open-source solution for robotics. It is more an acknowledgement of the complexity inherently connected with developing systems that perform in and interact with real environments.)

To remediate the deficiencies of the baseline solution, the adaptation layer has generally free access to the robot navigation. In particular, it can obtain estimates of the position and can sense whether the robot moves. Based on this, it can:

- manipulate the queue of locations to be visited (destinations)
- pause the robot and command the robot to move to any place on the map

Additionally, the adaptation layer on one robot may communicate with the adaptation layers of other robots to realize more complex adaptation strategies via cooperation.

The adaptation however comes with another set of problems, once we try not only to recover the robots from failures and deadlocks, but also optimize the overall performance of the system. Clearly, by reordering the locations to be visited and by transferring the responsibility of cleaning a place from one robot to another, the system can highly optimize itself. Theoretically, it can even get to a point when no collisions happen because robots exchange their destinations in such a way that they do not interfere. This is however subject to multiple problems, which can be regarded as additional adaptation challenges:

- The uncertainty in location makes planning not completely reliable.
- Communication range is limited, which means that robots in different rooms cannot communicate directly, but only through proxies (if present), which have to be located in the corridor close to the office entrances.
- The communication is subject to latencies and unreliability (due to interference) which makes it impossible for a robot to have an up-to-date knowledge of the global state of the system and disallows strong synchronization among robots.

2.2 Solution Comparison Dimensions

Having the adaptation logic in place, various metrics can be considered for evaluation and comparison of different adaptation strategies (*solutions* to ACRC). We list below metrics which we found useful in our experiments with ROS-controlled robots. Note that since ACRC contains random elements and non-determinism, the evaluation of a solution requires multiple simulation runs of ACRC and statistical evaluation (e.g. by statistical testing of sample means or quantiles).

[1] http://wiki.ros.org/

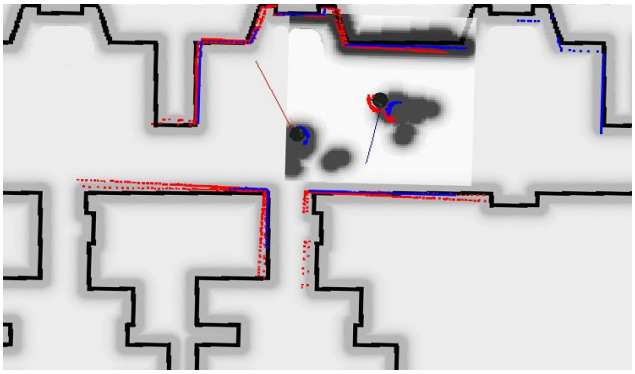

Figure 2. Robots' perception of the environment.

Time to complete all the tasks (i.e., visit and clean all locations assigned to the robots at the start) can be regarded as the basic metric when we assume that the evaluated solution is able to make the robots complete all their tasks. Our experience showed that this is more difficult than it appears to be. For evaluating partial successes, we thus suggest the following metrics.

Number of cleaning tasks that were completed. This covers situations when time limit for completion expires or when the system itself realizes that certain locations cannot be cleaned – e.g., if a robot gets stuck in a room entrance and any attempts to move the robot out of the way fail.

Total running time till system completes or gives up. This can be used as a metric complementary to the above one, to reward solutions which possess the ability to recognize that certain problems cannot be solved. It can serve to resolve ties in case two solutions are statistically similar (e.g. a statistical test cannot reject the hypothesis of the two solutions have the same average number of cleaning tasks completed).

3. TESTBED

The testbed provided as part of the artifact allows for easy experimentation with adaptation techniques and algorithms for the ACRC problem. It is built as a combination of ROS, the environment simulator and visualizer (Stage), network simulator (OMNeT++) and a component model for sCPS (DEECo). Details on the technical architecture is given in Section 3.3.

The testbed provides a simulation of a swarm of Turtlebots[2]. However, as it is developed on top of the ROS stack, it can be adapted to accommodate other ROS-controlled robots by configuring the Stage environment simulator to take into account different robots' dimension, their movement characteristics, sensors, etc.

3.1 User's Perspective

The testbed models the ACRC problem via DEECo component model (described in more detail in Section 3.2). In particular, it represents each robot as an instance of `CleanerRobot` component and provides its baseline behavior (i.e. the base-level subsystem [12]) in Java. The testbed provides a well-defined place in the component where the adaptation logic is to be plugged in (i.e. the reflective subsystem). Technically, this is done by introducing additional periodic processes to the Collector Robot component, and additional ensemble specifications (e.g. see Section 4.1).

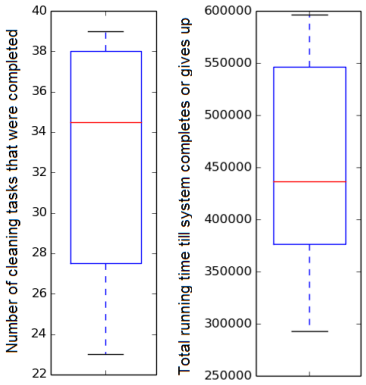

Figure 3. Boxplots of results from 10 experiment runs.

The actual ACRC simulation is configured by the number of robots and their initial positions. The testbed comes with one map that comprises a corridor and two offices. Custom maps can be provided as PNG files similar to the one shown in Figure 1.

The primary output of the testbed is the direct visualization of the scenario shown in Figure 1 (the visualizer itself comes with the Stage robot simulator – Section 3.3). Via it, the user can observe the movement of the robots at real time. Black lines represent walls and other obstacles impenetrable for the robots (i.e., the map provided to the testbed). The colored dots represent the robots. Also, it is possible to observe the robots' view of their environment – Figure 2. The red and blue dots represent the robots' perception about the walls/obstacles. In Figure 2, there are two robots represented as the bigger black dots; the red dots are associated with the first robot, the blue ones with the second robot. The grey areas next to robots have a higher cost for the planning algorithm of the robot (i.e., robots try to avoid them).

Further, the testbed comes with a script which computes statistics of the evaluation from the logs collected in multiple simulation runs. It generates boxplots of the results for the last two metrics defined in Section 2.2 (as in Figure 3).

3.2 Modeling Concepts for Decentralized Coordination

The robots behavior is developed using DEECo [2], which is a component model and framework for developing complex sCPS. DEECo is based on concepts of *ensemble-based component systems* (EBCS) (designed primarily in the scope of the EU FP7 ASCENS project[3]). In EBCS, a system is modeled as a set of dynamic cooperation groups of software components – *ensembles*. DEECo itself is an abstract component model, however it comes with two implementations – one in Java[4] (JDEECo) and one in C++[5] (CDEECo). In the artifact, we use JDEECo as we found Java easier for prototyping the components.

A component in DEECo is represented by its data (*knowledge* in EBCS) and its tasks (*processes* in EBCS). Figure 5 shows a code skeleton of the baseline implementation of the robot component in JDEECo. JDEECo-specific constructs are expressed using an internal domain specific language (DSL) defined via Java annotations. A component is defined as a plain Java class annotated with the `@Component` annotation. Component's knowledge is

[2] http://www.turtlebot.com/

[3] http://ascens-ist.eu/

[4] http://github.com/d3scomp/JDEECo

[5] http://github.com/d3scomp/CDEECo

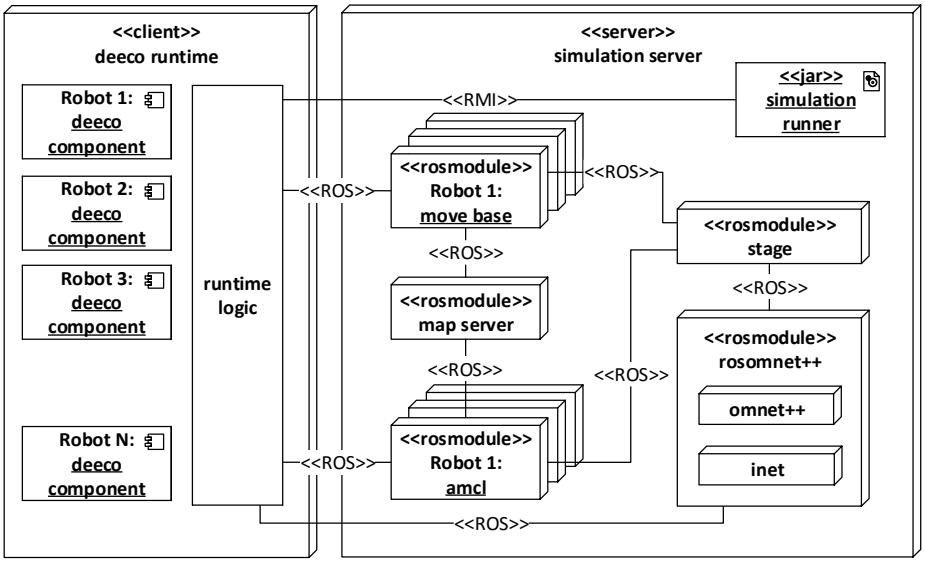

Figure 4. Testbed deployment diagram.

defined as Java class fields (lines 3-14 in Figure 5). Knowledge that is not supposed to be shared with other components via ensembles (as described below) is marked as `@Local`. Component's fields are manipulated by the component's processes (e.g. lines 16-33). Processes are defined as respectively annotated static Java class methods. Processes are either periodically executed or event-triggered (i.e., commonly as a reaction to knowledge change). This is determined by the annotation attached to the process.

Typically, processes involve sensing, computation, mutation of the component knowledge fields, and actuating. The signature of the process defines which knowledge fields are read/written (as in/out/in-out). Technically, the processes are scheduled by JDEECo runtime, which also takes care of thread-safe retrieval of component's knowledge to be used by a process and storing of the process results back in component's knowledge.

Figure 5 lists the processes defined in the baseline implementation of the cleaner robot as provided by ACRC. These are (i) setting the next destination, (ii) reading the position, (iii) reporting the status, and (iv) controlling the movement of the robot.

Communication between components is in DEECo modeled by *ensembles*. An ensemble dynamically determines which components are in the communication group via a membership condition. Topologically, an ensemble in JDEECo is a star featuring one *coordinator* and multiple *members*. The communication within ensembles is implicit, i.e., the ensemble defines an exchange method, which performs knowledge exchange among components grouped in the ensemble (i.e. copying data from a knowledge field of one component to a knowledge field of another component).

The baseline implementation does not involve any ensembles. However, ensembles are to be exploited for decentralized coordination of adaptation across several robots. This is demonstrated in Figure 6, where an ensemble for location exchange is given. It is established between robots which are close to each other and both of them are stuck. The ensembles are defined again as plain Java classes with annotations. The membership and exchange methods are periodically executed (with prescribed period – line 2) and their parameters specify the read/written knowledge fields of particular components (prefixes *coord* and *mbr* are used to identify coordinator and member role respectively).

```
1.  @Component
2.  public class CleanerRobot {
3.    public String id;
4.    public Position destination;
5.    public Position position;
6.    public State state;
7.    @Local public Long blockedCounter;
8.    @Local public Long noPosChangeCounter;
9.    @Local public Position oldPosition;
10.   @Local public List<Position> route;
11.   ...

12.   @Process
13.   @PeriodicScheduling(period = 500)
14.   public static void setDestination(
15.      @InOut("destination")
           ParamHolder<Position> destination,...)
16.   {...}

17.   @Process
18.   @PeriodicScheduling(period = 100)
19.   public static void  sense( @Out("position")
        ParamHolder<Position> position,
        @In("positioning") Positioning positioning)
20.   {...}

21.   @Process
22.   @PeriodicScheduling(period = 1000)
23.   public static void reportStatus( @In("id")
        String id, ...)
24.   {...}


25.   @Process
26.   @PeriodicScheduling(period = 2000)
27.   public static void driveRobot(
        @In("position") Position pos,
        @In("positioning") Positioning positioning,
        @In("destination") Position destination,
        @InOut("curDestination")
28.      ParamHolder<Position> curDestination,...)
29.   {...}
30. }
```

Figure 5. Model of ACRC baseline in JDEECo.

3.3 Technical Architecture

Figure 4 shows the architecture of the testbed. Technically, it is a merger of four main existing modules. The contribution of the testbed lies properly configuring them and bridging them by glue and synchronization code. The modules are:

- ROS Core – this module provides the basic software of the robot, implements the AMCL localization, navigation and low-level movement control of the robot.
- OMNeT++[6] – is a network simulator. It runs independently of ROS. We have implemented a bridge between ROS Core publish/subscribe mechanism and OMNeT++, which exposes the MANET transceiver as a ROS topic. This allows modules connected to ROS to communicate. OMNeT++ simulates the latency, physical range and interference of the communication based on robots' positions.
- Stage[7] – is a robot simulator, which controls the simulation. It connects to ROS Core and simulates sensors and actuators of the robot given the simulated robot position and the map of the environment.
- JDEECo – provides the component abstraction and concepts for decentralized coordination as described in Section 3.2. It abstracts ROS topics on location, navigation and exposes them to DEECo components to allow for adaptation. It further exploits the ROS topic on MANET-based communication (backed by OMNeT++) to implement inter-component communication via ensembles. JDEECo again runs independently of ROS and is synchronized with it by a bridge that we have developed as part of the testbed.

4. EVALUATION

4.1 Example Adaptation Logic

We complement the model problem specification and the testbed with an example adaptation logic as part of the model problem. It provides a comprehensive example of the modeling concepts (described in Section 3.2) and also serves as evaluation of the testbed to perform simulation of physical, mobility, networking and coordination concerns.

In the example adaptation, we tackle the problems described in Section 2.1 in the following way:

a) We introduce a process (on each robot), which periodically detects the situation when a robot is stuck. This is done by checking whether the robot is moving and whether the robot has a destination set. The robot that is not moving and wants to move is considered stuck.

b) If a robot is detected to be stuck, we select a random location from its queue of destinations and set it as its current one. This resets the navigation module in the robot and typically gets the robot to move. We monitor the outcome via the process described in (a) and repeat if no visible outcome is detected.

c) If another robot is stuck in close proximity (up to 1.5m), we establish and ensemble with it. Within the ensemble, one robot adopts the current destination of the other robot and vice-versa. This solves the (deadlock) situations when two robots meet in the office entrance and cannot proceed.

The strategy (c) is illustrated in Figure 6. Ensemble membership is defined on lines 6-12, and destination adoption is defined on lines 19-24.

4.2 Lessons Learned and Limitations of the Testbed

The experience with development of the testbed on top of ROS led us to several observations, which we believe are of general interest. We thus share them here.

Generally, a relatively big surprise was the overall immaturity of the frameworks. This most likely stems from the fact that ROS is primarily used as a platform for controlling a single robot at real-time. Though it has very flexible architecture, which allows running multiple robots within a single ROS system and allows connecting different environment simulators (e.g. Stage), the practice shows that these setups work out of the box only for trivial examples. Deploying multiple robots without careful configuration of the environment would make ROS or the Stage simulator crash. Similar story applies for OMNeT++, which is a mature and production-ready network simulator used in many applications. Nevertheless, when it comes to complex exercising of the mobile

```
1.  @Ensemble
2.  @PeriodicScheduling(period = 3000)
3.  public class DestinationAdoptionEnsemble {
4.    double MAX_DIST_M = 3.0;
5.    long BACKOFF_MS = 10000;

6.    @Membership
7.    bool membership(@In("coord.id") String
        coordId, ...) {
8.      return coordState == Block &&
9.             mbrState == Block &&
10.            !coordId.equals(mbrId) &&
11.       coordPos.distTo(mbrPos) < MAX_DIST_M;
12.   }

13.   @KnowledgeExchange
14.   void exchng(@InOut("mbr.destination")
15.       destination, ...) {
16.     if (now-lastAdoption.value)<BACKOFF_MS) {
17.       return;
18.     }
19.     mbrAdoptedDestinations.value
20.       .add(coordDestination);
21.     mbrDestination.value = coordDestination;
22.     mbrRoute.value.add(coordDestination);
23.     mbrBlockCnt.value = 0l;
24.     lastAdoption.value = now;
25.   }
26. }
```

Figure 6. Excerpt from example ACRC adaptation strategy.

ad-hoc network, the simulator again becomes very fragile and without careful configuration and patching, it crashes for no obvious reason. From this perspective, we believe that even without the DEECo abstraction layer, the pre-configured testbed we provide can save a couple of months of painful debugging.

Another class of problems comes from the fact that, though ROS has been used in simulations, it is not a discrete event simulator. It consists of a number of modules, which just run in wall-clock time. This means that (1) the simulation is non-deterministic, and (2) if extra care is not taken, the system crashes because the simulator, ROS, OMNeT++ and DEECo are not synchronized. We have

[6] http://omnetpp.org/

[7] http://playerstage.sourceforge.net/

solved this problem by introducing explicit synchronization at critical places, but still one has to keep in mind that this solution does not result in fully deterministic simulations.

Surprisingly enough, our experience with developing the sample adaptation logic has shown that the wall-clock timed simulation has certain advantages over a standard off-line discrete-event simulation. Since the system is live (and behaves as if the robots were moving in real time), one can watch the system as it runs, inspect the laser scans, etc. Additionally, it is possible to modify the system while it is running – e.g., a robot can be dragged by mouse to another location. While this is not important in classical batch simulations which focus on statistical comparison of different algorithms, it is very useful in debugging and especially in prototyping (which in fact is one of the primary goals of our test-bed and the reason why we equipped the testbed with DEECo abstractions).

5. ARTIFACT STRUCTURE

The ideas described in this paper are supported by an artifact available at http://d3s.mff.cuni.cz/projects/components_and_services/deeco/files/seams-2016-artifact.zip. The artifact contains the source code of the testbed, together with installation and usage instructions. Moreover, a pre-configured virtual machine image is included in order to enable rapid hands-on experience without the hassle of installing tons of libraries.

The artifact is formed by a single archive which contains all the necessary files. Instructions on how to use the artifact are located inside of the archive in index.html.

6. RELATED WORK

In this section, we briefly review three model problems/exemplars that have been contributed to the self-adaptive systems community repository [7]. This is an ongoing effort to provide benchmarks to evaluate new techniques against the state of the art, a popular strategy in other communities such as performance engineering (DaCapo suite [1], SPEC benchmarks [10]).

The automated traffic routing problem (ATRP) [3] is a model problem that can be used as benchmark for the evaluation of different self-adaptation mechanisms. ATPR features cars traveling on a map. Each car has a specific starting point, a specific destination, and a specific starting time. Each car has the goal to reach its destination by traversing the map, while respecting the speed limits on the streets. Problems arise due to conflicts between individual goals (e.g., all cars select the same street resulting in traffic congestion on the street), traffic accidents and road closures. ATRP can have solutions that are fundamentally different ranging from centralized to completely decentralized ones and generating answers that are optimal or sub-optimal. These solutions can be compared w.r.t. dimensions such as scalability, answer quality, robustness to sensor uncertainty, etc.

To evaluate and compare ATRP solutions, ADASIM has been proposed [3]. ADASIM is a Java-based discrete event simulator that simulates the execution of an ATRP solution on an ATRP instance. It provides configuration files for specifying the problem instance, built-in routing algorithms, traffic delay functions, filters for introducing measurement uncertainty, and Java interfaces that can be implemented to specify an ATPR solution. An event logging and analysis infrastructure is also provided. In summary, ATRP provides a vehicle suitable for experimentation with different self-adaptation strategies that try to resolve conflicts between goals of individual agents, prioritize between non-functional properties, and provide robustness to faults. Similarly, our model problem and testbed stand as a benchmark for self-adaptation mechanisms, but focuses more on run-time uncertainty and unreliable communication and coordination in sCPS.

Znn.com [9] is another model problem for self-adaptation. Znn.com is a news service that serves multimedia content to its customers. It is realized by a classical N-tier client-server architecture. The objective of Znn.com is to serve content while optimizing operating costs and keeping the response time bounded. Self-adaptation capabilities are needed in order for Znn.com to react to spikes on user load or other external changes while satisfying its objectives. In such cases, Znn.com can choose from a limited number of pre-designed adaptation decisions, e.g., switching the content served from multimedia to textual or incrementing the server pool size. While Znn.com is primarily suitable for comparing self-optimization solutions, our model problem and exemplar is more suitable for comparing solutions that focus on self-healing and survivability (robot unblocking, deadlock resolution) properties of sCPS.

Tele Assistance System (TAS) [11] is an exemplar for self-adaptation in the area of service-based systems. TAS aims to aid patient suffering from chronic conditions via tele-assistance. It is realized by wearable sensors measuring vital parameters and three remote services for data analysis, medication delivery, and ambulance dispatching in case of emergency. TAS comes with a number of generic adaptation scenarios. Each scenario consists of the type of uncertainty that warrants self-adaptation (e.g., service failure), appropriate self-adaptation actions (e.g., switching to equivalent service) and type of quality attributes (QoS) impacted (e.g., cost). For measuring the satisfaction level of each QoS, respective metrics are specified. A reference implementation of TAS [11] provides a convenient way of comparing self-adaptation solutions w.r.t. user-specified requirements in user-specified settings (instances of the generic TAS scenarios) by simulating them and analyzing the results with built-in graphical tools. While an excellent representative exemplar for self-adaptive systems, TAS focuses specifically on service-based systems, not CPS.

7. CONTRIBUTIONS

Responding to the pressing need for model problems and testbeds to evaluate the research ideas in the area of self-adaptive smart Cyber-Physical Systems (sCPS), we have presented ACRC, a model problem in the realm of sCPS that lends itself to a number of self-adaptation techniques that increase its self-healing, survivability, and self-optimization properties. It facilitates the process of trying out and comparing self-adaptation solutions to this problem. Our pre-configured testbed provides a starting point for experimental research. We hope that ACRC will help increase the awareness of the yet-to-be-addressed challenges in the exciting field of self-adaptive sCPS and drive further advances in the field.

8. ACKNOWLEDGMENTS

The work on this paper has been partially supported by the Charles University Grant Agency project No. 391115 and Charles University institutional funding SVV-2016-260331. This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie (StMWi).

9. REFERENCES

[1] Blackburn, S.M. et al. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *Proc. of OOPSLA '06* (2006), 169–190.

[2] Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M. and Plasil, F. 2013. DEECo: An ensemble-based

component system. *Proceedings of CBSE 2013, Vancouver, Canada* (Jun. 2013), 81–90.

[3] Jochen Wuttke, Yuriy Brun, Alessandra Gorla and Jonathan Ramaswamy 2012. Traffic Routing for Evaluating Self-Adaptation. *Proc. of SEAMS '12* (2012), 27 – 32.

[4] Kim, B.K. and Kumar, P.R. 2012. Cyber–Physical Systems: A Perspective at the Centennial. *Proceedings of the IEEE*. 100, Special Centennial (2012), 1287–1308.

[5] NIST 2012. *Cyber-Physical Systems: Situation Analysis of Current Trends , Technologies , and Challenges*.

[6] Salehie, M. and Tahvildari, L. 2009. Self-Adaptive Software: Landscape and Research Challenges. *ACM Transactions on Autonomous and Adaptive Systems*. 4, 2, May (2009), 1–40.

[7] Self-adaptive systems community repository: *http://self-adaptive.org/exemplars*. Accessed: 2016-01-22.

[8] Sha, L., Gopalakrishnan, S., Liu, X. and Wang, Q. 2008. Cyber-Physical Systems: A New Frontier. *Proceedings of the 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing* (Jun. 2008), 1–9.

[9] Shang-Wen Cheng, Bradley Schmerl. Znn model problem: *http://self-adaptive.org/exemplars/model-problem-znn-com/*. Accessed: 2016-01-22.

[10] SPEC benchmarks: *http://www.spec.org/benchmarks.html*. Accessed: 2016-01-22.

[11] Weyns, D. and Calinescu, R. 2015. Tele Assistance: A Self-Adaptive Service-Based System Examplar. *Proc. of SEAMS '15* (2015).

[12] Weyns, D., Malek, S. and Andersson, J. 2010. FORMS: A Formal Reference Model for Self-adaptation. *Proc. of the 7th International Conference on Autonomic Computing* (2010), 205–214.