Continuous Collaboration: A Case Study on the Development of an Adaptive Cyber-Physical System

Matthias Hölzl* and Thomas Gabor*
*Ludwig-Maximilians-Universität München, Germany

***Abstract*—The need to interact with complex environments that are often not well understood at design time makes the development of smart cyber-physical systems (sCPS) a challenging endeavor. We propose a set of practices and tools that support the design and implementation of sCPS using continuous collaboration—a development lifecycle and architecture to continuously incorporate data gained from the operation of the sCPS into the system. Continuous collaboration attempts to harmonize three interlocking feedback cycles: refinement of the system design by the developers, autonomous evolution of agents in the sCPS, and feedback from the evolving system to the developers. To support the process we introduce tools and techniques that we have found helpful to realize continuous collaboration: The HADES/Hexameter platform, extended behavior trees and the teacher/student learning pattern.**

I. Introduction

Smart cyber-physical systems (sCPS) are powerful tools that are ever more widely deployed. However, the physical world is well known for its non-deterministic and occasionally baffling behavior. Developing sCPS that behave intelligently—or at least safely and appropriately—during their interactions with the outside world is therefore challenging, and traditional software-engineering techniques are not particularly well-suited to these systems.

Instead of fighting against the uncertainty and non-determinism inherent in the physical world, it seems sensible to embrace them. We therefore propose a novel development approach, called *continuous collaboration,* which is based on the teacher/student system architecture presented in [1] and can incorporate these properties while still leaving the developers in control of the system's overall behavior.

To this end, we will first introduce a scenario investigated by the ASCENS project [2]: a swarm of autonomic robots performing a rescue operation. We then present the first novel result of this paper, our *continuous collaboration* approach to modeling and building software for sCPS, and show how it can be used to develop software for the robot swarm. This approach blends traditional software engineering methods with techniques from machine learning and evolutionary computing to obtain systems that respect constraints imposed by their designers while exploring and finding novel solutions inside this frame. In section IV we provide a short analysis of the performance of a system built according to this approach which represents the second novel result. Section V presents related work. Finally, section VI sums up the experiences gained from the case study and provides an outlook to future developments on the continuous collaboration approach.

II. The "Robot Rescue Force" Example

The increasing miniaturization of hardware has greatly contributed to the feasibility of swarm computing: instead of employing relatively few, high-powered devices a swarm consists of many simple nodes. While each node has only limited computational power and physical capabilities, the nodes in the swarm cooperate to solve problems that are beyond the faculties of individual nodes.

One interesting application area for swarms is disaster relief and rescue operations. In these situations, the environment is extremely hazardous and non-predictable. To allow humans to stay outside the immediate danger zone it is desirable to have robots that can operate autonomously. In addition, available human rescue forces are typically stretched thin and operating under severe time constraints; autonomously operating robots could alleviate some of this pressure. Since robots operating in these environments may be damaged, e.g., by crumbling buildings, it is desirable to not rely on a few large and expensive robots, but rather on many simple and cheap ones.

In this paper we will use the following example: An industrial complex has been damaged; workers have been trapped in several buildings and need to be rescued. It is expected that parts of the complex will further collapse while the rescue operation is in progress. In this paper we ignore the difficulties inherent in working with real robots and look only at the problem of navigation in an unknown environment for which the robots can only be partially prepared by the developer of the CPS. In particular we focus on adapting as quickly as possible to changes in the environment, and on minimizing the overhead for internal organization and planning.

III. Continuous Collaboration

In this section, we present a software architecture and development approach that can be used to tackle the previously mentioned challenges. The collaborative aspects of sCPS development permeate all phases of the software lifecycle; we therefore start the discussion with the presentation of a communication and network infrastructure (subsection A). We then introduce the language we use for the specification of agent behavior in subsection B and show how it lends itself to the development of systems that integrate individual and collective learning and adaptation (subsection C). Finally, we discuss some aspects of the overall process of *continuous collaboration*, a development pattern based on using feedback gained from the running system to continuously adapt and improve a system's operation throughout its life

978-1-4673-7088-2/15 $31.00 © 2015 IEEE
DOI 10.1109/SEsCPS.2015.12

cycle (subsection D). Our implementation of the rescue robot case study relies on the techniques presented in this section; the performance of the respective implementation will be examined in section IV. The software for this experiment can be downloaded from the "Academia" project (`https://github.com/hoelzl/Academia`).

A. The HADES/Hexameter Platform

As the core piece of our software infrastructure, we wrote HADES ("HADES's A Discrete-time Environment Simulator"), which provides a framework for the communication of the robot controllers with their environment. It defines the notifications passed to the agents and how they can access their sensors and actuators, thus providing a common world model to all agents. HADES can work both as a server running a simulation of the environment or as meta-layer on top of another simulator or actual robot hardware; therefore we can use the same robot controllers to drive experiments either in an abstract environment (see section IV) or in a fully simulated physical world. To achieve the latter, we have integrated HADES with the ARGoS swarm-robotics simulator [3], which is capable of simulating large swarms of robots with realistic models for physics, sensors and actuators.

As HADES is implemented in Lua, a lightweight scripting language designed for easy embedding into applications written in C or C++, it can easily be embedded in other projects and is well-suited to running on limited hardware.

HADES uses a protocol named *Hexameter* for all of its network communication, which is a general-purpose protocol based on the primitives of the SCEL modeling and programming language. SCEL was developed within the ASCENS project and is "a new language specifically designed to rigorously model and program autonomic components and their interaction" [4]. Basing Hexameter on SCEL has the distinct advantage that the communication behavior of components using Hexameter can easily be modeled in SCEL and that these models are compatible with the techniques and results developed in the ASCENS project.

As front-end libraries are available in various languages, HADES provides a flexible way to connect external software components as well as to port the whole system to different simulated or physical platforms.

B. Extended Behavior Trees (XBTs)

Behavior trees (BTs) are a technique for describing agent behaviors that was introduced to allow designers to script the behavior of non-player characters in computer games. In its simplest form a BT is a tree containing *actions* that are executed by an agent as leaf nodes, and *choice nodes* and *sequence nodes* as internal nodes. Choice nodes allow the choice between different courses of action whereas sequence nodes allow sequential execution of actions. Each BT is periodically activated by the control loop of its containing system (adopting a term commonly used in game engines, we say it is *ticked*) and returns a result indicating that its execution was either successful, that execution failed, or that it is still running and needs more time to continue. A choice node succeeds when any of its child nodes succeeds and fails when all of its child nodes have failed. A sequence node fails whenever one of its children fails and succeeds when all its children succeed. Both choice and sequence nodes return a running status whenever one of their children does.

Many implementations of BTs exist; most of them extend the basic node types with nodes such as *decorators*, internal nodes having a single child and modifying the execution behavior of that child, e.g., by "negating" the result (from succeeded to failed and vice versa). Despite their simplicity, BTs allow surprisingly complex behaviors to be specified, but they suffer from a number of well-known limitations, for example: (1) Every action performed by the BT is immediately executed so that possibilites for off-line planning are limited. (2) It is difficult to implement state-based behaviors using BTs, since the child nodes of each node have to be specified in the tree which leads to clumsy specifications if different states require different behaviors.

To address these issues we have introduced *Extended Behavior Trees (XBTs)* in [1]. XBTs extend behavior trees in several ways, among them:

- The nodes return an indicator about the reward they obtained to their parent in addition to the execution status.
- A state object is threaded through the activations of the nodes. All operations on the environment have to be mediated by the state, e.g., by being implemented as methods of the state object. States can be *virtualized* so that operations on the state simulate the expected behavior but do not change the real environment.
- There is a node type *external choice node* that is similar to choice nodes but calls a function to generate its child nodes whenever it is ticked and not already running.

These extensions of behavior trees make it easy to specify state-based behaviors and they permit reinforcement-learning and offline-planning nodes. In addition, the operational semantics of XBTs is defined by a translation into SCEL and therefore node types that execute their children concurrently are easy to define.

Fig. 1 shows an XBT for a simple rescue robot. The topmost node is a choice between four alternatives: The leftmost subtree checks whether the robot is in one of its home locations. In that case, it executes its child node, a sequence node that first updates the robot's data and then drops off the victim. If this alternative fails, either because the robot is not in a home location or because it is not carrying a victim, the choice node proceeds to the second subtree, checks whether there is a victim in the current location and picks up the victim if possible. If this branch of the tree also fails and the robot is carrying a victim, it picks a home location and moves towards this location, otherwise it picks a new location where it expects to find a victim to rescue and moves towards that location.

Returning a result status of "running" is similar to yielding in a coroutine: execution of the currently running branch of the XBT is suspended and control returns to the control loop; when the XBT is ticked again, it will resume execution

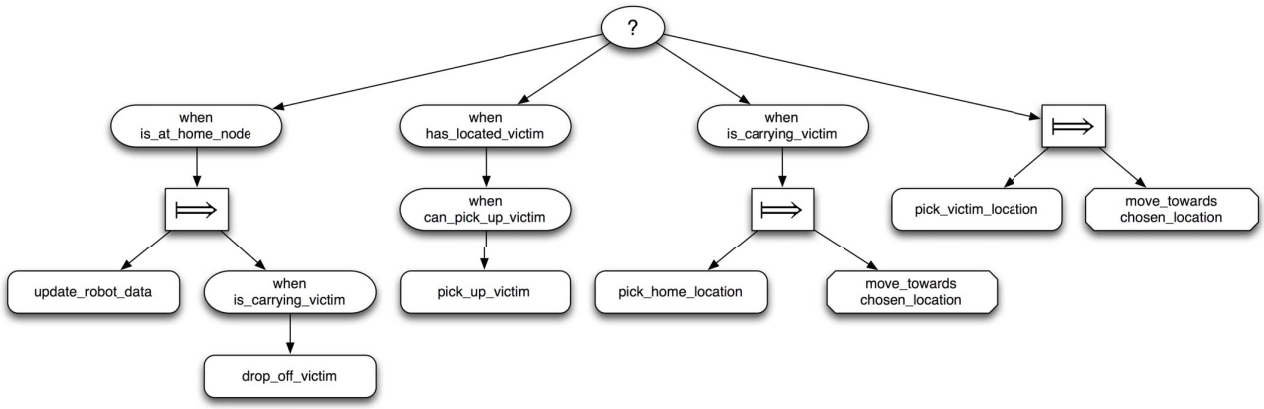

Fig. 1. XBT for a rescue robot

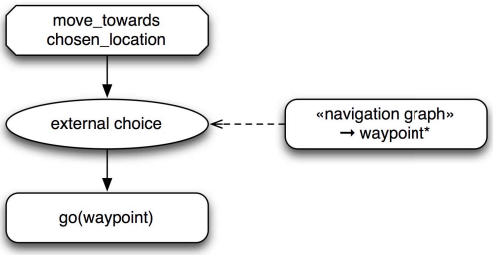

Fig. 2. State-dependent XBT node

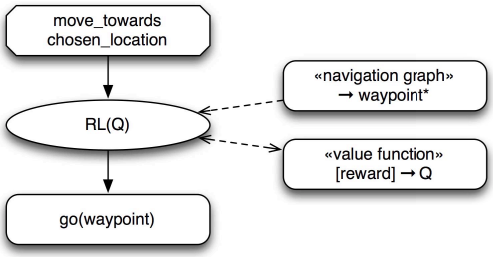

Fig. 3. Reinforcement-learning XBT node

starting with the suspended node. The possibility to fail and continue with the next choice leads to a convenient handling of situations where the outcomes of actions are not certain.

Moving towards a target location is one example of state-based behavior: the direction to a target depends on the current position of the robot. Fig. 2 shows how external choice nodes can address this situation: The state of the navigation graph provides a sequence of waypoints that are possible targets for the next move of the robot. The external choice node tries to perform a go action to each of these nodes in turn, until one action succeeds. By providing a sequence of waypoints, the system achieves a limited amount of error tolerance: when some paths are blocked, the robot tries to find alternative routes. But this solution is rather limited: the different waypoints may lead to greatly different rewards or costs, and the experience of the robot from previous rescue missions is not used to improve future behavior since there is no feedback from the XBT back to the navigation graph.

In the next sections we will show how robots using the XBTs in Figs. 1 and 2 can cooperate to adapt to changes in the environment. However, it is also easy to incorporate individual learning into XBT-based controllers to add additional flexibility: Fig. 3 shows the movement XBT for a robot that autonomously learns the navigation decisions for unknown terrain. This navigation system uses a RL(Q)-node, an external choice node that performs reinforcement learning using a Q-function. This node receives the set of all possible next waypoints from the navigation graph and uses an estimate of their quality (provided by the Q-function) to choose the order in which they are tried. Since the available Q-function is generally only an approximation of the real values obtained from the environment, the RL(Q) node uses the reward obtained by executing the go(waypoint) child node to update its estimate of the Q-function. In addition, the RL(Q) node occasionally performs *exploration moves* to waypoints that are not optimal according to the current estimate in order to obtain a better estimate. As can be seen by this example, it is straightforward to add individual learning behaviors to any choice node in an XBT.

C. Teacher/Student Learning

Even if the environment or requirements of an sCPS are not completely known at design time, the system designers can usually provide at least a rough idea about algorithms that are well-suited for accomplishing the goals of the system, and about the structure of the system's activities. In addition, domain-specific quality criteria have to be elucidated in the requirements phase, e.g., in the form of a utility measure. Thus, it seems logical to include these rough guidelines into the sCPS not only as a default mechanism or a constraint on the system's behavior, but as the basis of a learning process. More concretely, we can express the behavioral requirements

in the form of an XBT that defines the structure of the system's behavior. Each choice in the XBT can be hardwired, if the designers are confident about the correct choices or they can be specified as, e.g., RL(Q) nodes which each component in the system can optimize based on its experience. In addition, the simple structure of XBTs allows multiple agents in the system to exchange information about beneficial choices for certain nodes *even if they share only certain subtrees of the XBT* and not the whole behavior. Therefore the sCPS can adapt its behavior by incorporating the shared experience of its components and thus hopefully improve its performance in situations where its programming is not well suited for the task at hand. In order to do so, we have designed an internal communication pattern that is able to allow this adaptation process with little overhead.

We identify certain components of the sCPS as *teachers*, whose job it is to provide domain knowledge to other agents, called *students*; their main job is to interact with the environment according to the plan provided by the teachers. However, the distinction is not always strict: it is possible to think of scenarios where agents fulfill both roles at once as well as of scenarios featuring a strict separation, e.g., because students are autonomous robots whereas teachers are large stationary computers. Teacher/student learning as a design pattern is more thoroughly covered in [1]; the rest of the section will focus on its use as framework for adaptive behavior.

In the simplest implementation of the "robot rescue force", there is a teacher present at the home base, which interacts with any robotic agent passing through. The teacher contains the knowledge of the original plan of the industrial complex (before the site was possibly damaged) in the form a graph with weighted edges representing the estimated traveling cost and can thus compute the seemingly optimal routes for the robots to travel through the navigation graph in order to reach the nearest victims. The table of shortest paths between on-site locations (i.e. nodes in the navigation graph) is then passed on to the students residing in the home base. After they have used this table to navigate the site as well as possible, hopefully locating a victim and returning it to the home base in the process, the teacher retrieves a record of their actions and their actual consequences regarding damages and rewards from the students, which it can use to improve its own model and subsequently teach an advanced version of the navigation table to its students.

However, this process of information distribution is rather simplistic: On the one hand, the transmission of possibly very large amounts of data from teachers to students and vice versa may not be feasible under the specific circumstances, due to limitations in used hardware, especially bandwidth. On the other hand, interference with the environment might distort message content or break the communication protocol. In general, it can be assumed that communications between agents may end up rather noisy. In this context, bandwidth limitations can be viewed as a special form of noise as well, i.e. coherent parts of a transmission are lost entirely. But instead of trying to take corrective action against problems in message transmission, a learning algorithm can actually take advantage of it: Agents using reinforcement learning techniques rely on a certain amount of "exploratory" behavior in which they perform actions they consider non-optimal to improve their estimate of the utility of various states. Unless the communication channel is too noisy, we can set up the data transmission so that natural noise adds a variation to the plans enacted by the agents that is similar to the one required for exploratory actions, and thus leads to an increased exploration of the solution space that can, if desired, be compensated by reducing the deliberately introduced exploratory behavior. This gives rise to a natural form of mutation in individual solutions between the single students. In addition, the structure of XBTs makes it straightforward to not exchange complete plans or strategies but only decisions for individual nodes in the XBT, leading to a kind of cross-over recombination. In summary, due to the structure of the learning process mimicking the actual CPS architecture, environmental interference, which is prone to occur at the point of agent communication, can be mitigated or even exploited by teacher/student learning.

Selection can also be implemented in the teacher/student model and occurs whenever a vicinity features multiple teachers trying to share their recommendations with the nearby students. Since the students can only do one action at a time, they have to decide between different solutions they are offered at some point in time. While it is entirely possible for students to remember multiple recommendations and decide on one just in time for each respective action, or to combine them if the type of action permits such an operation, in our implementation, the choice between multiple recommendations offered by different teachers happens at the moment of teaching: Due to previous experiences with the recommendations of a specific teacher, the student develops a relationship of trust with that teacher. Since the teacher passes on an expected reward alongside with its recommendations, the student can compare that expectation with the reward it actually gained by enacting the recommendations and thus increase or decrease its trust towards that teacher accordingly. The higher the trust a student has for a teacher, the more likely it is to accept that teacher's recommendations and replace its current navigation table with them. Thus, if a teacher can provide good recommendations more often, it is more likely to distribute further recommendations among the students of the sCPS. Likewise, teachers usually providing wrong information will be almost ignored by the students over time.

These two simple elements (i.e. mutation and selection) make up a simple evolutionary algorithm improving on the individual action plans. Note however, that the plans that make up the population of the evolutionary process are constantly used for robot control, even while they are still evolving. When combined with robots executing said plans in parallel and without a single point of control (as in our case), this approach is called "embodied evolution". Employing this technique, we can provide additional adaptive abilities to the sCPS without bringing in the overhead of a full-fledged engine for learning or evolutionary algorithms in particular, but just by

choosing the right communication pattern and exploiting its natural properties. However, since the overall behavior of the controller is still constrained by the structure of the XBT, designers can control which parts of the system take part in the evolutionary process, which bounds are imposed on the evolution, and which behaviors remain fixed.

D. The Continuous Collaboration Pattern

The teacher/student paradigm brings further advantages when considering the design process of an sCPS: as it provides a natural way of using multiple different knowledge sources in conjunction, it aids the developer in crafting the sCPS's emergent behavior from smaller strategies. This also allows the developer to include knowledge gained from single-agent versions of the problem, which are usually much easier to analyze computationally. The presence of a respective teacher promoting at least roughly feasible solutions to its student can thereby influence the behavior of the whole group of agents.

This also provides an easy interface for later adjustments to the sCPS: By adding a teacher, the system will adopt that teacher's recommendations if they prove to be promising, but will quickly learn to ignore that teacher otherwise, thus not causing a too big disturbance in the productive run.

From a software engineering point of view, the teacher/student dynamic provides a continuous interface for handling the sCPS's emergent behavior throughout the system's life cycle: New behavior can be implemented by adding a new teacher or multiple teachers accordingly. The sCPS, however, will only actually opt for this behavior, if the first probabilistic tests (i.e. the few students randomly picking new teachers not yet assigned any trust value over presumably quite trustworthy, well-known teachers) show at least no detrimental decrease in fitness. Furthermore, the analysis of the trust teachers gained from their students provides a way of gathering feedback about the apparent efficiency of employed strategies (at least from a robot's point of view) and can direct developers to promising points of improvement. Similarly, feedback from the outside (from domain experts, e.g.) can also be incorporated by adjusting the available teachers and their respective recommendations. Therefore, the teachers provide a middle layer mediating between the software developer and the emergent system behavior enacted by all agents, aiding in abstraction and separation of concerns.

The inherent locality of most communication inside an sCPS operating according to the teacher/student pattern also allows for a high level of scalability: Since teachers only operate with students in their vicinity, the size of the group of agents can easily be increased while keeping the teacher:student ratio constant without running into any complexity pitfalls. From a development point of view, this means that once the basic infrastructure is set up, teams can easily be split between different parts of the behavioral logic.

The development process that controls ongoing agent behavior through the mediation of the teachers, expressed through both adjustment from the outside and gathering feedback from the inside of the agent group, is what we call *continuous collaboration*.

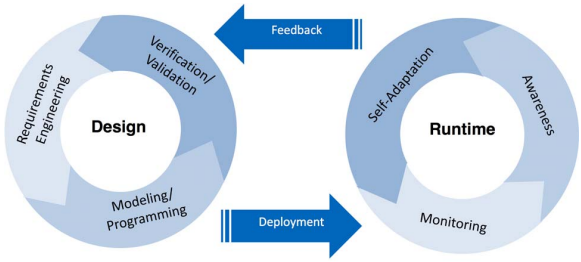

Fig. 4. The Ensemble Development Life Cycle (EDLC)

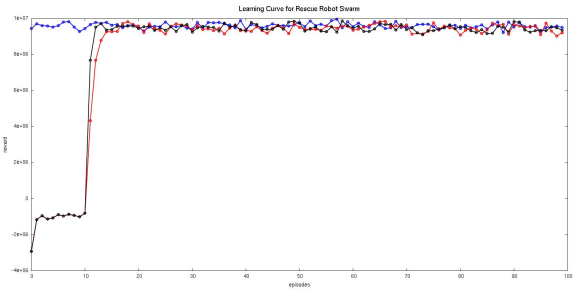

Fig. 5. Performance of the rescue robots over time, static case

A life-cycle model for this kind of highly iterative and feedback-intensive development has been proposed by the ASCENS project as *Ensemble Development Life Cycle (EDLC)* [5]. The EDLC features two feedback loops, processed at design time and run time, respectively. They are connected by deployment (passing results from design time to the run time system) and execution feedback (passing run time information back to the development cycle), resulting in a third feedback loop, see Fig. 4. *Continuous collaboration* can be seen as an instance of the EDLC where the teachers have a dual role, as both part of the awareness and self-adaptation phases of the runtime cycle, and as mechanism for deploying new or updated behaviors and providing feedback about the system's operation to the developers. Therefore teacher/student learning provides a common interface for both online and offline adaptation.

IV. Experiments

To evaluate the teacher/student learning pattern we have implemented the robot rescue scenario using the tools and techniques described in this paper. While lack of space prevents us from discussing the case study in detail we want to give at least a short qualitative overview of the main results.

The robots use the XBT in Fig. 1 with state-dependent navigation but no individual learning to operate in a navigation graph with 200 nodes connected (initially) by 3960 randomly generated (directed) edges.

Robots can traverse edges as well as pick up and drop off victims. Each edge has a weight that determines the cost for

traversing that edge; when a robot drops off a victim at a designated home node it receives a reward. The experiment is organized into 100 episodes. At the start of the first episode all robots are located at the home node. They start executing their XBTs, therefore they obtain a plan from a teacher and then start exploring the graph and rescuing victims. When a robot returns to the home node it provides a log of its actions and the obtained rewards to the teacher from which it obtained its plan; it can then request a new plan from this or another teacher. In each episode each robot performs 250 ticks of its XBT. After each episode, each teacher can update its internal model of the world; the end of the episode has no effect on the robots (i.e., they stay where they are and start the next episode in the state in which they finished the previous one).

Fig. 5 shows the performance of three teachers for a robot swarm consisting of 50 robots. The blue curve (starting at the top left corner of the diagram) is a teacher with perfect information about the environment following an optimal policy, i.e., in the average case this represents the best possible performance in this environment. This teacher does not take into account information provided by the students and does not suggest exploratory moves to its students. The (red and black) curves starting in the lower left corner show the performance of teachers with imperfect information that learn from the information provided by their students and whose students perform exploratory moves. Initially the graph known by the red and black teachers differs significantly from the actual environment both structurally and from the expected rewards: 10% of the edges of the real graph are missing, 5% of the edges that are not present in the real graph are present in the teachers' graphs, and the cost of each edge in these graphs differs significantly from the real cost.

Each learning teacher uses a dynamic programming algorithm to compute the best paths between any two nodes in its current model of the world and teaches that model to robots when they request new data. We therefore call these red and black teachers DP-teachers. They differ only in how eager they are to exploit the information in their model in the face of errors in the model (i.e., discrepancies between their predictions and the rewards observed by their students).

The imprecise initial model of both DP-teachers is reflected by their very poor performance during the first episodes. Since the results reported by the students are significantly different from their estimates, both DP-teachers suggest the maximum exploration rates to their students, which causes the students to essentially perform random walks. (We set the maximum exploration rate to 80% so that the students always try to exploit the plans to some extent. In our experience this leads to quicker convergence than pure random walks.) However, as can be seen from Fig. 5, after only 10 episodes, the DP-teachers rapidly approach the performance of the optimal teacher. Since the DP-teachers incorporate data from multiple robots, their plans converge much more quickly than the plan any student could develop on its own.

This first experiment shows that teacher/student learning can converge to the optimal solution in a static environment. The more realistic and interesting case is the behavior of the system in dynamic environments. We simulate two different scenarios:

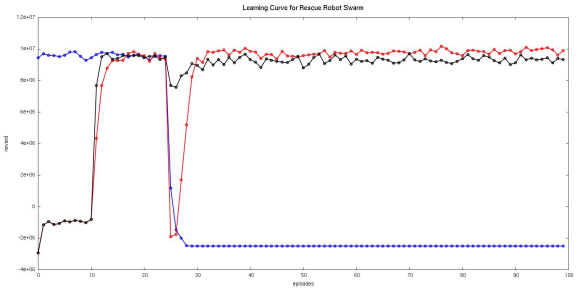

Fig. 6. Performance of the rescue robots over time, "catastrophic" change

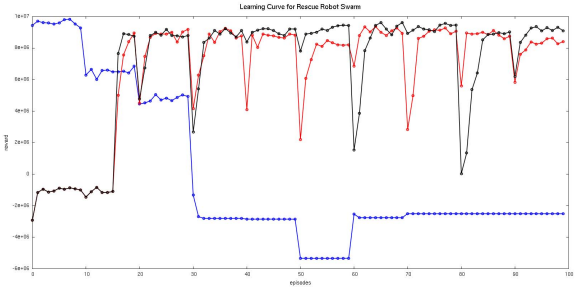

Fig. 7. Performance of the rescue robots over time, gradual change

Fig. 6 shows the same experiment as Fig. 5, but after 25 episodes a "catastrophic event" happens in the environment and the structure of the graph is changed significantly: 50% of the existing edges are destroyed, 10% of the previously unconnected nodes are connected, and the weights of the remaining edges are modified. The black teacher immediately reacts to this change and updates the behavior of its students so that only a slight decrease in their performance can be observed. The previously optimal teacher fares very badly since its map no longer reflects the reality of the new environment and it does not take any new information about the environment into account. The performance of the red teacher drops for a short period, but after 5 episodes its performance has recovered and it retains this performance thereafter.

The third experiment simulates an environment that degrades not in one catastrophic event but periodically; the results for this experiment are shown in Fig. 7: 15% of the existing edges are destroyed, 2.5% of the unconnected nodes are connected and the weight of the existing edges is modified every 10 episodes. As expected, the performance of the blue (non-learning) teacher decreases significantly as the graph starts to deviate from its map. The performance of the DP-teachers decreases, sometimes significantly, at the beginning of each episode, but they recover quickly and remain near the optimal result level for a large percentage of the time. If both DP-teachers are teaching simultaneously, the system performance can further be improved by having students learn from the teachers in proportion to the expected reward but

providing information about the rewards to both teachers. Using this strategy the average performance of the robots is only slightly below the maximum value of both DP-teachers and the "valleys" in the performance are smoothed out.

To summarize, a teacher/student swarm with DP-teachers achieves nearly optimal performance across all three scenarios after a short learning period. If its students choose their teachers in proportion to the expected rewards the swarm can eliminate some of the variance introduced by the learning and exploration process while still responding quickly to changes in the environment. Since the teachers integrate the results of all their students, the learning process converges quickly.

V. Related Work

Many languages for modeling and implementing behaviors exist, but none of them seems to be a perfect fit for the requirements of sCPS: UML activity and state diagrams [6] are commonly used to describe the behavior of embedded systems, but they are not particularly well-suited for specifying goal-directed behaviors or integrating learning or planning mechanisms into a system. Programming languages extended with constructs for incorporating learning or planning, such as ALisp [7], are very flexible and expressive but it is often difficult to understand how planning, learning and adaptation integrate with the preprogrammed behavior, and the resulting programs are difficult to understand for non-programmers.

BTs have been implemented in a number of game engines [8] and are increasingly used for applications in robotics and avionics [9], [10].

Embodied evolution is defined in [11]. A more in-depth discussion as to why this approach is reasonable for scenarios like the "robot rescue force" is given in [12], including special characteristics of "embodied evolution" and possible pitfalls.

Many mechanisms for learning and adaptation have been proposed; [1] contains an overview and further references of reinforcement learning approaches; [13] introduces many techniques for multi-agent learning and communication.

VI. Conclusion and Future Work

We have introduced a case study on the development of an adaptive CPS using continuous collaboration in the form of the teacher/student approach to learning. This conceptual approach is supported by XBTs for behavioral specification and the HADES/Hexameter platform as runtime. The experimental results show that the system achieves competitive performance even in static scenarios and excels in situation where frequent adaptation is necessary.

There are many interesting directions for future work. On the theoretical level, there are obvious connections to evolutionary game theory, distributed reinforcement learning and artificial evolution. In particular, the application of replicator dynamics and the investigation of evolutionary stable strategies to guide the introduction of teachers seems promising.

We have restricted the discussion in this paper to one sub-problem suggested by the scenario while ignoring many of the difficulties present in the interaction with a physical environment. As next step we plan to extend the experiment to operate on robots simulated in ARGoS. In this context it is also interesting to investigate how existing algorithms for robot localization, in particular for Simultaneous Localization and Mapping (SLAM) [14], can be integrated into the continuous collaboration approach.

From a software engineering point of view, an approach like continuous collaboration raises many questions, for example, which requirements, analysis and design processes are compatible with the ongoing integration of run time data and knowledge into the system. Another important aspect is how verification and validation can be combined with continuous collaboration, and which constraints on the system's behavior can be guaranteed.

Acknowledgment

This research was supported by the European project IP 257414 (ASCENS).

References

[1] M. Hölzl and T. Gabor, "Reasoning and Learning for Awareness and Adaptation," in *Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project*, ser. LNCS, M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, Eds. Springer, 2015, vol. 8998.

[2] M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, Eds., *Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project*, ser. LNCS. Springer, 2015, vol. 8998.

[3] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, M. Birattari, L. M. Gambardella, and M. Dorigo, "ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems," *Swarm Intelligence*, vol. 6, no. 4, pp. 271–295, 2012.

[4] R. D. Nicola, D. Latella, A. L. Lafuente, M. Loreti, A. Margheri, M. Massink, A. Morichetta, R. Pugliese, F. Tiezzi, and A. Vandin, "The SCEL Language: Design, Implementation, Verification," in *Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project*, ser. LNCS, M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, Eds. Springer, 2015, vol. 8998.

[5] M. Hölzl, N. Koch, M. Puviani, M. Wirsing, and F. Zambonelli, "The Ensemble Development Life Cycle and Best Practises for Collective Autonomic Systems," in *Software Engineering for Collective Autonomic Systems: Results of the ASCENS Project*, ser. LNCS, M. Wirsing, M. Hölzl, N. Koch, and P. Mayer, Eds. Springer, 2015, vol. 8998.

[6] Object Management Group, "UML Specifications," http://www.omg.org/spec/, last accessed: 2015-02-26.

[7] D. Andre, "Programmable reinforcement learning agents," Ph.D. dissertation, University of California at Berkeley, 2003.

[8] I. Millington and J. Funge, *Artificial Intelligence for Games*, 2nd ed. Morgan Kaufmann, 2009.

[9] A. Marzinotto, M. Colledanchise, C. Smith, and P. Ögren, "Towards a unified behavior trees framework for robot control," in *2014 IEEE Int. Conf. on Robotics and Automation, ICRA 2014, Hong Kong*. IEEE, 2014, pp. 5420–5427.

[10] P. Ogren, "Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees," *AIAA Guidance, Navigation and Control Conference, Minneapolis, Minnesota*, pp. 13–16, 2012.

[11] R. A. Watson, S. G. Ficici, and J. B. Pollack, "Embodied evolution: Distributing an evolutionary algorithm in a population of robots." *Robotics and Autonomous Systems*, vol. 39, no. 1, pp. 1–18, 2002.

[12] G. Karafotias, E. Haasdijk, and A. E. Eiben, "An algorithm for distributed on-line, on-board evolutionary robotics," in *Proc. 13th Ann. Conf. on Genetic and Evolutionary Computation*, ser. GECCO '11. New York, NY, USA: ACM, 2011, pp. 171–178.

[13] Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. New York, NY, USA: Cambridge University Press, 2008.

[14] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2005.