Multi-agent Simulation Approach to Development of Applications for Decentralized Tactical Missions

Antonín Komenda, Michal Čáp, Michal Pěchouček

{*komenda,cap,pechoucek*}*@agents.felk.cvut.cz*
Department of Computer Science and Engineering,
Faculty of Electrical Engineering,
Czech Technical University in Prague

***Abstract*—The development of control algorithms for tactical missions is being impeded by the significant gap between the way the artificial intelligence (A.I.) algorithms have been designed and validated and the way the robotic applications for (high-fidelity simulations) of tactical missions are being developed. On the one hand, we have low-level robotic simulators (or even robotic field testing). On the other hand, we have synthetic – usually mathematically defined – environments used for the design and formal testing of A.I. algorithms, e.g. randomly generated problem instances, synthetic graph structures, logical structures, regular grids, and similar.**

In this work, we are proposing a development process and a related software toolkit helping to narrow this gap. We use the simulation-aided development approach and tailor it towards the domain of tactical missions. The process is demonstrated on a specific application scenario, employing a general software toolkit Alite to glue together and adapt a number of A.I. algorithms, originally designed as highly abstract.

I. INTRODUCTION

In recent years, we have been witnessing an intensive development and deployment of various robotic systems. One of the fastest-growing application domains for robotic systems are the Intelligence, Surveillance, Target Acquisition, and Reconnaissance (ISTAR) military missions performed by remotely controlled robotic assets. Currently, these robotics assets are controlled and coordinated exclusively by human operators. The scalability of such approach is clearly constrained by the limits of human perception and the limits of inter-human interactions, similarly as in the other fields, where the human element was superseded by computerized systems.

To address these constraints, there have been large investments in successive introduction of autonomous multi-robotic systems. In such systems, the robotic assets use the distributed artificial intelligence to coordinate their actions and cooperate with each other. We will use the term *Decentralized Tactical Missions* to denote the class of problems, where multi-robotic teams carry out tasks in ISTAR missions [9], support disaster relief operations or assist humanitarian missions [16], [10].

The fundamental challenge associated with the multi-robotic application development is the deployment, validation and verification of the developed algorithms on the real hardware. Conducting experiments on real-world robots is expensive both in terms of time and money. To reduce such costs, a *simulation* of the target system can be introduced. On the one hand, the experiments in a simulated world have the advantages of the reproducibility, direct control over the simulated world, and usually also the efficiency of experimenting (one can conduct batch experiments). On the other hand, the fundamental drawback of the simulated-world experiments is that the accuracy of the results depends on the fidelity of the world model employed by the simulation. Since the computational complexity of the simulation grows as the function of the fidelity of the underlying world model, high-fidelity simulations are often impossible to achieve due to their prohibitive computational complexity.

A. Problem Addressed

Nowadays, we can identify a significant gap between the environments typically used by the researchers in the theoretical A.I. community and the environments used by the developers of multi-robotic intelligent applications. On the one hand, we have synthetic environments used for design and formal testing of different kinds of A.I. algorithms (e.g., randomly generated graphs, regular grids, etc.). One the other hand, we have high-fidelity real-world-like robotic simulators (or even mixed-reality simulations with real robots) used for a pre-deployment evaluation of the proposed multi-robotic application.

In our approach, based on the simulation-aided design of multi-agent systems methodology [15], the key motivation is to bridge the gap between the theory and the practical application of the existing A.I. algorithms. Under such motivation, we take existing A.I. algorithms and starting from a low-fidelity, highly abstract synthetic environment we incrementally decrease the *level of simulation abstraction* to eventually arrive to a high-fidelity real-world-like environment. During each step, we adapt the used algorithms and validate the overall function of the system.

To enable such an iterative process of algorithm design and validation, we have to implement the algorithms on a simulation platform that is *modular and flexible enough* to provide seamless *testing of the algorithms working on different levels of abstraction*.

B. Problem Domain

To demonstrate the key ideas of the proposed approach based on simulation-aided development, we need a suitable example domain. As our main motivation is to design and test primarily algorithms of distributed artificial intelligence for

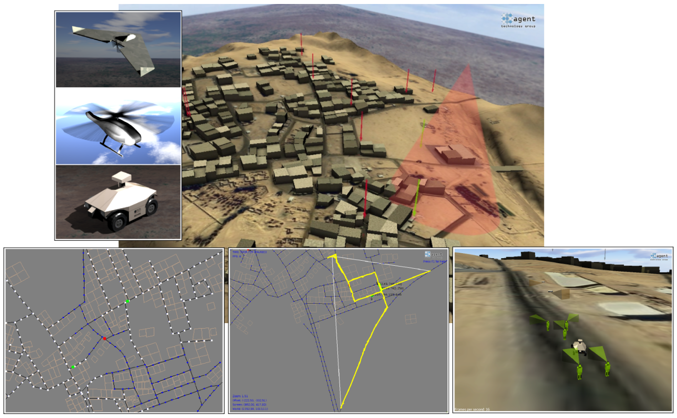


Figure 1. A visual impression of a simulated environment for Decentralized Tactical Missions.

teams of autonomous robotic assets, we will choose an ISTAR military mission domain. Such a tactical domain gives us a wide range of possibilities regarding both the available robotic assets and the tasks the assets are expected to perform. As we can see in Figure 1, the environment comprises a village, the surrounding uneven landscape, models of the available robotic assets, friendly persons and adversarial persons.

We consider two main groups of assets: unmanned ground vehicles (UGVs) and unmanned aerial vehicles (UAVs). Additionally, on all levels of fidelity, we distinguish Conventional Take-off and Landing (CTOL) UAVs and Vertical Take-Off and Landing (VTOL) UAVs as their movement models fundamentally differ. To enrich the set of possible tasks for the robotic assets, we also simulate the behavior of friendly (blue) and enemy (red) forces forming teams and convoys. The tasks carried out by the robotic team include patrolling of allied convoys, capturing evading adversaries, low-level formation maintenance, team support by observing local area, or wider area surveillance.

In the following section we will explain the simulation-aided iterative development process used to design, implement and validate the above-presented multi-robotic application. The details about the mission scenario, implementation and the A.I. techniques applied will be described in Section III. The Section IV concludes the paper with final remarks.

II. DEVELOPMENT PROCESS

The main idea of the presented development process is the following. At the beginning, we employ the classical theoretical A.I. approach and design the desired algorithm in a synthetic environment, using general mathematical structures such as graphs and grids. However, right from the start, we perform the experiments within the framework of the *target* simulation system. This means that the interfaces between the developed algorithm and the simulated environment must be general enough to allow straightforward redeployment of the algorithm to higher fidelity simulation environments. Further, the synthetic environment should also define sufficiently general interfaces to allow future integration with higher fidelity simulation environments.

The requirement for general interface should not interfere with the function and the internal principles of the developed algorithm. In this step the simulation system acts purely as a validation environment for the developed algorithm respecting all its simplifying assumptions. After validating and verifying the algorithm in its pure form, we can iteratively replace the environment model (or possibly other parts of the simulation, e.g., the mode of time evolution) and re-validate the algorithm in an environment containing more aspects of the target environment, i.e., having a lower level of abstraction. Occasionally, after the abstraction of the simulation environment has been decreased, the tested algorithm has to be conservatively adapted. A *conservative adaptation* of an algorithm is an adaptation that preserves all the desired mathematical properties (e.g. soundness, completeness, etc.) for the price of possibly newly added domain-specific constraints on the validity of these properties. The final sum of such adaptations results in a theoretically-backed algorithm applicable in highly detailed simulated environments. The mathematical properties of the algorithm stay valid under the constraints introduced by the applied conservative adaptations.

In the next subsections we will describe the simulation approaches we use for design and validation of the target algorithms.

A. Simulation-aided Development

The simulation-aided development (SAD, as described in [15] and [8]) is based on an iterative process of an approximated validation using testbeds of increasing fidelity. The goal of the process is a successful, cost-efficient deployment of the application on the target system, typically a hardware platform. The iterative process of the application development is based on the feedback from approximated testing. The approximation is based on two dimensions: *level of abstraction* (how much is the target system simplified) and *scope of abstraction* (which parts of the target system are simplified). In result, the initial system consisting of highly abstract algorithms is iteratively transformed with increasing level of detail in each step into a practical system deployable on a hardware platform.

As we stated earlier, the main objective of our work is to design and experimentally evaluate various decentralized algorithms for coordination of multi-agent teams. This leads to the development of applications employing such algorithms using the principles of SAD. In contrast to [15], our final goal is not to deploy the algorithms on a hardware platform, but on a high-fidelity simulation. This objective results in three basic requirements on the simulation system.

Firstly, the simulator must be highly configurable to allow for high flexibility in terms of a) simulation experiment structure (number of agents, various types of agents, different initial conditions, etc.) and b) executed scenario storyboard, i.e., the mission to be executed. This requirement is related mainly to the scope of abstraction in SAD, as we have to be able to easily reconfigure parts of the simulation and switch between different environment models and different models of simulated entities.

Secondly, the experimental validation requires the results from a large number of simulation runs. To ensure properties

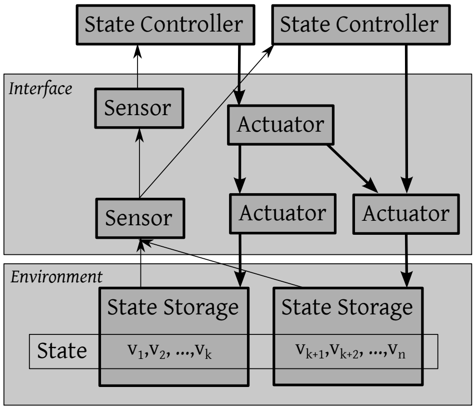


Figure 2. An example of a simulated environment, described by state variables $v_1, ..., v_n$ separated into two state storages. The state controllers (e.g., agents) perceive and act in the environment through a set of sensors and actuators respectively. One of the sensors and one of the actuators (the top two) acts as a high-level abstraction for low-level ones (e.g., autopilot actuator on top; yoke and pedals actuators on bottom).

of the tested algorithms during the adaptation process, the simulation platform has to facilitate straightforward construction of experiment suites. Batch experiments represent a basic technique to a validate a wide spectrum of problem instances and experimentally prove the desired properties of an algorithm. Conditioned and dynamic experiments can be used for search of pathological or otherwise important problem instances and related results.

Finally, the simulator has to allow simulation on different levels of details of the simulated environment and the simulated entities. The last requirement is closely related to the level of abstraction in the SAD approach, as the algorithms have to be allowed to work seamlessly among various levels of abstractions to enable automated testing. Automated testing ensures that the properties of an algorithm hold on all levels of abstraction (analogically to automated testing as used in classical software engineering).

B. Environment Modeling

A fundamental part of the simulation platform is a model of the virtual environment, which comprises the description of the *simulated state* and the *state controllers* animating the simulated world. The state controllers are driven by a *time management* component.

We represent the state of environment as a set of special containers called *state storages* (see Figure 2) . Each state storage is responsible for holding a specific part of the current state, i.e., all the state storages together constitute the full description of the current state of the environment. The partitioning of the simulated state into the state storages can be chosen arbitrarily, however the two most commonly used approaches to the state partitioning are the following: a) over the entity types or b) over the data types. The former uses one state storage per simulated entity type, i.e., a state storage contains the set of all state variables for all entities of one type (e.g., `CarStorage`, `HelicopterStorage`, `StreetStorage`). The latter is based on a data-type describing the state variables (e.g., `GraphStorage`, `KeyValueStorage`, `BTreeStorage`). In this case, one state storage contains all state variables of the same data structure and utilizes common properties of such structures, e.g., a `KeyValueStorage` can provide algorithms for hash-based caching, which can be utilized both for key-value storage of entity properties (size, weight, current fuel status) or key-value storage of an area weather status (keys represent area codes, values current weather conditions).

State controllers constitute the functional part of the environment, together describing the whole mechanics of the environment. The state controllers interact with the state of the environment indirectly through a set of universal interfaces called sensors and actuators. A sensor is an interface through which a particular part of the environment state can be read. Analogically, an actuator is an interface used to change a part of the environment state. Sensors and actuators are the only components permitted to directly access the state storages.

There are no *a priori* restrictions on the controllers and the controlled state, i.e., a controller can be a mechanism simulating physical laws of the environment (e.g. application of the gravity force to all simulated entities having mass), a simple reactive algorithm (e.g. simulation of swarm systems), or a highly deliberative algorithm (e.g. cognitive cooperating agents). Elements of the environment without any controllers are fixed in their initial state. These are e.g., the shape of the landscape, buildings, bridges etc.

In the tactical mission environment, the sensors and actuators are of different levels of complexity. There are basic sensors informing the controlling agents about their position in the simulated world (simulation of a on-board/personal GPS). The basic visual sensor simulates perception of other simulated entities in close proximity. The complex visual sensor emulates a system for automated friendly-or-foe detection using 3D algorithms to simulate visual occlusions caused by buildings and topology of the map.

To enable high-level control of UGVs, which abstracts away from the physical reality of the environment, an actuator for discrete-time movement of simulated ground vehicles on a street graph can be used. To enable various levels of abstraction, the high-level control algorithms use low-level actuators to steer the cars between waypoints on the street map (e.g., junctions) based on a state-of-the-art technology for simulated physics. Such an actuator supports not only simulated continuous motion of the entity in space, but also discrete motion on the graph-based representations. Such an approach to the design of simulated environment led to significant reduction of implementation, as well as debugging cost of the individual experimental scenarios on different abstraction levels. Moreover, it allowed us to implement a simulation model employing event-based time management. Unlike discrete time ticks or turn based time management methods, the event-based simulation allowed us to decouple the simulation time from the real-world “wall” time. The main advantage of this approach is that the time periods containing no simulation events can be skipped and thus the simulation runs execute significantly faster.

The requirement to implement aircraft performing close-up tracking of mobile targets, such as adversaries and cars, resulted in a need to incorporate aircrafts reactively controlled using low-level actuators, be it conventional fixed-wing planes (CTOLs), or helicopters (VTOLs). Such UAVs are able to change their flight trajectory in a reaction to changes of movement patterns performed by the ground target. In the case of fixed-wing aircrafts, which cannot stop in mid-air, this problem results in a need to perform relatively complex flight patterns, such as various types of loops over the target. Together with a need to implement a fine-grained physical dynamic feedback control of helicopters respecting a realistic model of their physical movements, this led to a requirement to adapt the simulator to a much finer grained time resolutions. In effect, reactive CTOL actuators use yaw, pitch and velocity as parameters and limits on minimal and maximal values, while VTOL actuators uses cyclic and collective rotor blade tilt for the main rotor and tilt for tail rotor using a simplified dynamic model of a VTOL. The higher level actuators such as a straight-flight autopilot and a waypoint autopilot make use of the lower level actuators and offer a high-level interface for more abstract control algorithms.

C. *Simulation Assurances*

While abstract mathematical algorithms are well analyzed and strongly statistically validated on experiments, it is not so easy to run (and debug) replicable experiments in complex, high-fidelity robotic simulations with lots of dynamic unpredictable behaviors of the entities and emergent behavior phenomena. In our approach, the important aspect of simulation development is to maintain the reproducibility of simulations with the increasing level of detail. Large-scale simulations involve various aspects of non-determinism which can lead to non-reproducible simulation runs. Such factors include parallel and random processes, as well as limitations of the underlying hardware, such as CPU scheduling or memory swapping, etc. To ensure reproducibility of experimental runs, we carefully considered and implemented the concept of *in vitro* simulation. That is, a simulation which controls all the aspects of the modeled system, or carefully accounts for those, which were abstracted away from. In particular, this means that the simulator has to have an ability to suspend and later resume the simulation process. Furthermore, it should have an ability to speed it up, or slow it down in response to e.g., resource utilization of the underlying hardware, so that race conditions and different results of process scheduling do not affect the simulation outcome. Finally, the random processes involved in the simulation must be also under the control of the simulator so that the same sequences of random events are generated in two independent runs of the same simulation.

The need to execute large numbers of reproducible experiment runs turned out to hinge on the speed of simulation run execution and ability to make the runs deterministic on demand. To tackle this issue, we departed from the exclusive model of centralized discrete time ticks and implemented *event-based simulation mechanism* [1]. This allows the system to disrespect real-time constraints of the wall clock ticking mechanism and run the simulation as fast as possible given the available computational hardware resources (memory and CPU). However, at the same time the resulting simulator still features the ability to run at real-time simulation speed for demonstration purposes. Additionally, we used simulator enabling complete synchronization of the simulated processes and thus facilitated high level of control over the simulated environment.

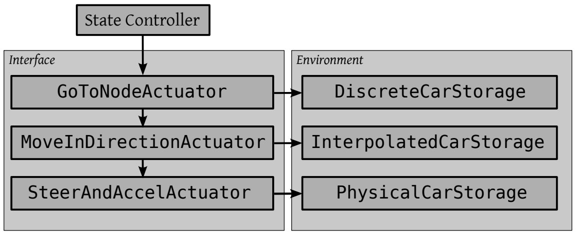


Figure 3. State storages and related actuators for description of car models on three levels of abstraction.

D. *Example of a Multi-level and -scope Abstractions*

The model based on state storages, universal sensors, universal actuators, and loosely coupled controllers offers a valuable property critical for the SAD approach. The property is that the presented model is highly flexible, as it allows a programmer to add and remove simulated entities easily (scope of abstraction). Further, it supports easy switching between the different types of simulation modes (level of abstraction).

For instance, we can define three types of abstraction for car entities used in the simulation and represent them by three separate storages `DiscreteCarStorage`, `InterpolatedCarStorage`, and `PhysicalCarStorage`. The first one defines the current state of a car by a node on the street graph. The second one enriches the by-node state with a position vector (x, y) representing the position of the car on a 3d mesh representing the ground surface. The last abstraction extends the state further with a description of a fully dynamic state comprising position (x, y, z), velocity $(\dot{x}, \dot{y}, \dot{z})$, acceleration $(\ddot{x}, \ddot{y}, \ddot{z})$ and the rotational components $(\varphi, \theta, \psi), (\dot{\varphi}, \dot{\theta}, \dot{\psi}), (\ddot{\varphi}, \ddot{\theta}, \ddot{\psi})$. To control the state stored in these state storages, we can use three actuators `GoToNodeActuator`, `MoveInDirectionActuator`, `SteerAndAccelerateActuator` (listed in an order reflecting the controlled state storages). One can implement an actuator to control the respective state storage directly, but it is also possible to implement an actuator to control storages indirectly through other actuators. In practice, such coupling will result in an algorithm that recursively translates higher level control to lower level control. For example, the way-point car actuator will control the car using the following control sequence: `GoToNodeActuator` $\rightarrow$ `MoveInDirectionActuator` $\rightarrow$ `SteerAndAccelerateActuator` (see Figure 3). As we can see now, we can interchangeably use any of the presented state storages as long as the controlling algorithm uses only the top-most actuator, i.e. `GoToNodeActuator`. In effect, we can design a high-level algorithm controlling a car only on node-to-node basis using `GoToNodeActuator`, but we can immediately test it in all prepared levels of abstraction (discrete, interpolated, physical).

We also define simulated entities representing the ground troops. Here, we create only two levels of abstraction represented by two state storages `DiscreteTroopStorage` and `DirectedTroopStorage`. The first level of abstraction is similar to `DiscreteCarStorage` (representing the position of a trooper in terms of street graph nodes), the latter describes the ground position and the direction (x, y, φ) of a trooper. We create a `WalkToNodeActuator` and `MoveAndTurnActuator`. We cannot reuse `GoToNodeActuator` in place of `WalkToNodeActuator` since the car actuator uses a different control logic to simulate the movement (although the input parameters and the results are identical for both the actuators – both the cars and the troopers move from one node to another – for the car, the duration of the movement can be computed from the engine power, for the trooper the duration of the movement can be, for instance, a function of the weight of the carried personal gear). From this point, we have a separate component for a car and a separate component for a trooper in the model of environment (analogical to the scope of abstraction in SAD). In a simulation run, we can use these components separately (just cars or just troops) or we can mix them together (e.g. troops following a car). Moreover, we can mix different levels of abstraction of both components (for instance, an interpolated car representing a convoy is followed by physically simulated cars representing UGVs accompanied by troops having position and direction representing the support squad protecting the convoy against the discrete adversaries blocking junctions on the street map).

III. APPLICATION SCENARIO

After proposing an approach to the development of multi-agent applications for decentralized tactical missions using simulation-aided development process, we present a specific application scenario in which the application was demonstrated. The application is a multi-agent simulation of a heterogeneous cooperative mission with opponents taking place in a dynamic environment. This section presents a detailed description of the mission, implementation details of the underlying system and a summary of algorithms used to control the behavior of agents. In particular, we will emphasize the role of different levels of abstraction used during the design and evaluation phases of the development process.

A. Tactical Mission

The mission takes place in a desert village surrounded by a hilly landscape. The village is described in terms of a number of static and dynamic objects. There are three types of static objects: buildings, bridges and a 3D mesh representing the ground. All the static objects act as obstacles for the dynamic objects and cause occlusions for the visual sensors. Additionally, there are virtual static structures: a street graph representing a navigation map of the village and forbidden zones representing the areas, where the dynamically simulated entities are not allowed to be (e.g. vicinity of the buildings, cliffs, edges of the bridge etc.). These virtual structures can be sensed by the entities, but unlike the physical obstacles, the agents can ignore them.

All the dynamic objects in the environment are denoted as *simulation entities*. A simulation entity is a simulated embodiment with a related controlling agent. In our simulated environment, there are no dynamic objects not deliberately controlled (e.g., moving obstacles, falling objects, etc.). The simulation entities can be divided into three main groups: air vehicles, ground vehicles, and simulated persons. There are three types of air vehicles: Aesir Vidar VTOL UAV, Saab Skeldar VTOL UAV, and Procerus CTOL UAV. We have two types of ground vehicles: MDARS UGV and a generic army cargo truck. And finally, the simulated persons can represent the allied troops (blue forces) or the adversaries (red forces).

The mission to be fulfilled is the evacuation of a VIP hostage from a safehouse in the center of the village and escort of the VIP to an extraction point at the end of the village. During both the ingress and the regress phase of the mission a highly valuable target (red forces) can be spotted. If such a situation occurs, the team (allied cargo truck and blue forces) splits and a part of the troops has to capture the evading target. There are unknown adversaries (red forces) operating in the village, who can endanger the members of the allied team. These have to be spotted as soon as possible to minimize the risk of attack against the team. The robotic support team (Vidars, Skeldars, Proceruses, and MDARS) autonomously provides backing to the ground troops by performing wide and close surveillance of the area, street and junction covering and others.

B. Implementation

The above presented multi-agent application has been implemented using an in-house software toolkit Alite, which allowed us to follow the principles of the simulation-aided development (SAD) approach.

Alite[1] *['eılaıt]* is a software toolkit simplifying implementation and construction of (not only) multi-agent simulations and multi-agent systems. The objectives of the toolkit are to provide a highly modular, flexible, and open set of functionalities defined by clear and simple APIs supporting rapid prototyping and fast implementation of multi-agent applications, mainly focusing on highly scalable and complex simulated environments. The guiding principles underlying the Alite design are i) modularity, so that the system does not commit a developer to a specific definition of concepts such as *agent*, *environment*, etc. and ii) composability, so that the various components of the toolkit can be put together in a rapid and flexible manner. In result, Alite can be seen as a collection of highly refined functional elements providing clear and simple APIs, allowing a programmer to put together relatively complex multi-agent simulation scenarios rapidly. In following lines, we explain the main characteristics and distinguishing features of Alite.

Alite agents have access to composable interfaces to the environment (sensors and actuators), while their internal decision-making process is not bound to any *a priori* philosophy. Additionally, they can make use of various types

---

[1] http://agents.felk.cvut.cz/projects#alite

of communication middleware interfaces allowing a developer to model various types of intra-agent communication (synchronous, asynchronous, peer-to-peer, broadcasting, multicasting, etc.). Further, Alite comes with libraries including various types of planners (reactive, deliberative) and multi-agent solvers (e.g., task allocators, solvers for distributed vehicle routing problem, etc.).

By its compositional nature, Alite provides means for both rapid prototyping, as well as high-level of elaboration tolerance of the implemented systems. E.g., once a simulation, or a functional multi-agent system is put together from various components, application-level customizations and proprietary domain-specific mechanisms, it is very easy to replace one stock planner, or multi-agent solver by another one, as far as they share the underlying assumptions for their use.

Alite addresses the problem of MAS platform resilience in the face of the need to incorporate various *a priori* unknown future requirements by variability in composition of functional elements. The number of possible combinations allows for construction a wide spectrum of structurally different multi-agent applications. This feature distinguishes Alite from the pre-designed frameworks such as [5], [17], [2]. As multi-agent application's requirements evolve, the requirements on the agent platform itself are changing. Alite does not provide "a single platform for all", but rather offers an efficient way to build a platform that fits the specific needs of the MAS application under development. The application can make use of one or more functional elements available in Alite toolkit. As of writing this paper, Alite provides the following packages:

- `common-event-queue`: a general implementation of a temporal event queue and temporal events (can be used for event-based simulations, agent message queues, etc.).
- `common-entities`: a general description of any entity in the system. An entity is defined only by its identity, i.e. name (represent agents, simulated embodiments, etc.).
- `common-capability-register`: a general implementation of a simple register of possible capabilities provided by entities (usable for directory services, register of simulation components, etc.).
- `communication`: a component providing communication interfaces and message transport layers (includes direct and asynchronous message transport, protocol abstraction, abstraction of communication modes, etc.)
- `initialization`: a component defining basic interfaces for initialization scripts and configuration (includes a Groovy-based config-reader)
- `environment`: a component of interfaces defining basic elements for creation of simulated worlds (includes state storages and base classes for implementation of sensors and actuator).
- `simulation`: a component providing a basis for event-based simulations (based on `common-event-queue` extended with temporal control).
- `visualization`: a set of components for visualisation of simulation outputs (includes 2D visualization, 3D visualization based on JME[2], wrapper to Google Earth[3], and others).

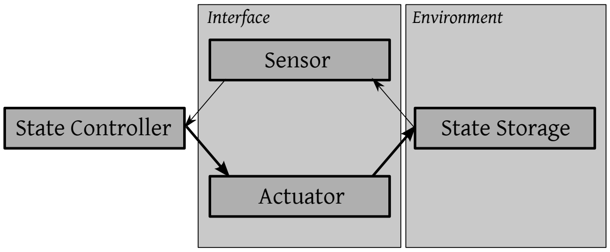


Figure 4. A full control loop in a typical Alite simulation architecture.

From the evaluation of basic multi-agent algorithms, it is just a small step to large-scale multi-agent simulations. Most of the general-purpose multi-agent platforms offer no support for implementation of complex simulated environments On the other hand, the simulation-oriented platforms lack support for implementation of complex agent behaviors and communication protocols. Alite stays in between these two approaches enabling an application developer to implement a multi-agent simulation platform targeting both mentioned aspects. Classical approach to multi-agent modeling as introduced in [7] and implemented e.g. in [17] incorporates simulation into the multi-agent system as a special agent. The simulation agent represents the simulated environment, entities and their interactions with the environment. The agents control their simulated bodies in the environment transparently using inter-agent communication. The simulation agent is responsible for the consistency of the simulated environment and synchronization of the entities. The reasoning processes of the individual agents run in separate, independent threads, the architecture is therefore suitable for parallelization and real-time simulations. On the other hand, the large-scale multi-agent simulation platforms such as Mason [13] or NetLogo [6] facilitate construction of large environments consisting of micro-behaviors of thousand individual (usually simple rule-based) agents that give rise to complex macro-behaviors.

Alite simulation adopts the *in vitro* principle, which represents a compromise between the two presented approaches. Classical simulation architecture is driven by the agent point of view (i.e. the agents live in the platform and simulation is merely one of the agents). The *in vitro* multi-agent simulation architecture is driven by the simulation itself (similarly to the large-scale simulation platforms). The agents' reasoning processes, the agents' bodies in the environment, function of actuators and sensors – everything is controlled by the simulation. Thanks to *in vitro* design, one does not have to explicitly distinguish between the behavior of an agent (i.e. agent's brain) and the behavior of an entity (i.e. agent's body). This approach allows a programmer to control all parameters (even those that cannot be controlled in the classical architecture, such as the computational power available to the agents' reasoning processes, the characteristics of the communication links between the agents, etc.). The simulation can be fully

---

[2] http://jmonkeyengine.com/
[3] http://earth.google.com/

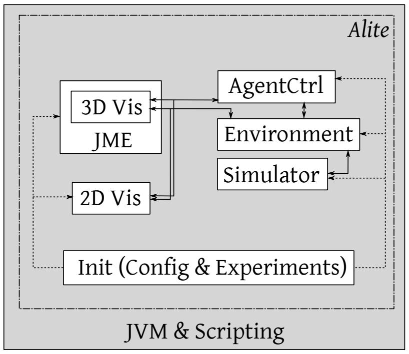


Figure 5. High-level overview of the simulation system for Decentralized Tactical Missions utilizing Alite toolkit.

deterministic, featuring simulated, controlled non-determinism if needed. Finally, the *in vitro* design prevents the simulation from being affected by the disruptive events occurring on the host computer, such as unbalanced processor load, unevenly distributed computation power to agents, etc.

Simulated environments in Alite consist of building blocks introduced in Section II-B: state storages, actuators, sensors, agents (i.e. the agents reasoning algorithms implemented merely as a specific type of Alite entity) and an event queue representing the time management component. A typical full control loop (see Figure 4) consists of `sensor` → `state controller` → `actuator` → `state storage` → `sensor` → . . . cycle. The state controller can be any Alite entity (i.e., an agent, a reactive controller, etc.) representing behavior of an element in the environment (e.g. a pilot agent, a traffic light, wind, a growing three, etc.).

The power of Alite's loosely coupled design has been tested during the construction of the multi-agent simulation system targeting the proposed domain of distributed tactical mission. A number of custom-designed domain specific components integrated with the Alite infrastructure components enabled us to transparently combine i) highly abstract multi-agent game-theoretical algorithms synthesizing strategies for patrolling of the allied forces and pursuit of intelligent evading targets, ii) the team member behavior generation based on continual planning, plan repairing and formation movement techniques, and iii) complex environment simulation, such as physics of rigid-body models based on JBullet physical simulator[4].

Our previous Alite-backed multi-agent application for the domain of multi-agent cooperation and coordination in complex urban environments has been presented in [19]. Thanks to Alite's highly modular architecture, there were minimal implementation overheads during the implementation of extended behavioral models employing special agent-oriented programming languages [18].

Based on the experience from our work with Alite, the final architecture of the system was designed as depicted in Figure 5. The agent control (`AgentCtrl`) component represents the agent's decision making algorithms. Agents act in an `Environment` that consists of entity-type state storages (`VidarStorage`, `SkeldarStorage`, `MdarsStorage`, etc.) using the respective sensors and actuators utilizing the multi-level and multi-scope abstraction principle (see Section II-B and Section II-D). The dynamics of the `Environment` is implemented using event-based full control loops (see above) backed by the event queue, together forming the main part of the `Simulator`. Further, the state of the environment (in particular, the positions of the entities) is visualized by `2D` and `3D Vis`-ualizators. Some of the important mental states of the agents' are also visualized (e.g. intentions, plans, an agent's prediction of the future, etc.). Finally, all the presented components of the system are initialized by the `Init`-ialization component, used to setup the entire experimental infrastructure via flexible configuration of the experiment suites (as proposed in Section II-A). Alite is written in Java language and other JVM-compatible languages (particularly Groovy[5] and Clojure[6]).

C. Algorithms and Evaluation

To demonstrate the proposed development process, the following section will discuss the design, verification and validation procedure of four A.I. algorithms that have been employed in the example evacuation mission.

1) Adversarial planning: patrolling of mobile targets: The protection of the ground team against the attacks from the adversaries is one of the objectives of the evacuation tactical mission. The protection was carried out by a small team of aerial vehicles. For the patrolling vehicle, it is vital not to execute a predictable movement strategy. If it acts predictably, the opponents could optimize their behavior against such strategy and attack the convoy in the worst timepoint, e.g., when the patrol just left the convoy it protects. The solution is based on randomized strategies, which maintain a certain average frequency of visits of each protected ground team. The algorithm computes optimal strategies for protecting the mobile targets in adversarial environments. The basic underlying assumption driving the research was that the opponent is able to observe the patrol and capable to attack in any moment when the target convoy is unprotected. Given a map of an urban environment, positions and plans of the convoys and a mobility model of opponent units, the specific goal was to find the optimal randomized strategy for the patrol, which minimizes the probability of attacks on the protected teams. More information about the underlying game-theoretical algorithm for patrolling can be found in [3], [4].

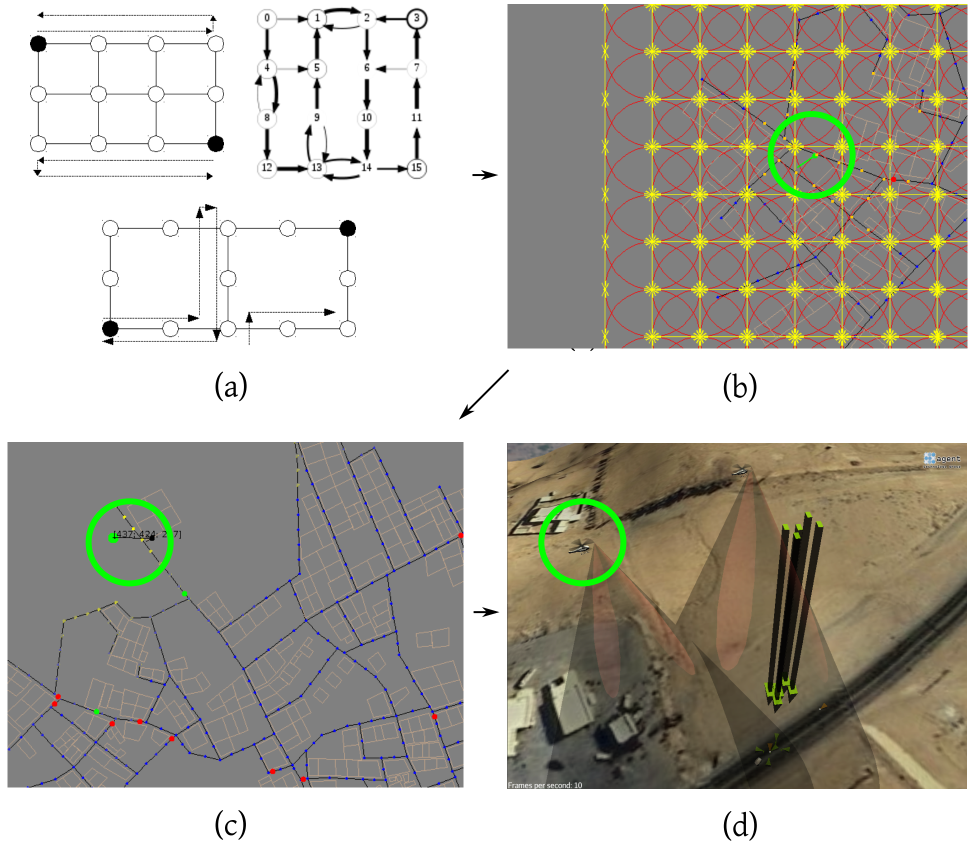
The design, development and validation of the algorithm had four steps (see Figure 6). In the first step, (a) the algorithm was analytically designed, based on state-of-the-art solutions, and the optimality guarantees were shown on synthetic graph structures. In the next step, (b) the algorithm was verified on targets traversing an urban area based on a graph representing topology of a real village. The patrolling assets were simulated as idealized models of a CTOL UAV using maneuvers from a discrete tessellated grid. Afterwards, (c) the idealized model of a CTOL airplane was replaced by

---

[4] for more information on the physics simulation see JBullet (http://jbullet.advel.cz/) – a Java port of Bullet Physics Library (http://bulletphysics.org)

[5] http://groovy.codehaus.org/

[6] http://clojure.org/



(a) (b) (c) (d)

Figure 6. Levels of abstraction used for design, verification and validation of the algorithm solving the problem of mobile target patrolling: (a) discrete movement on a synthetic graph structures, (b) discretized movement of CTOL airplanes using tessellated maneuver pattern, (c) movement based on a simplified dynamic model of Skeldar VTOL UAV, and (d) an integrated scenario where a VTOL UAV patrols ground tactical teams in the final high-fidelity simulator.

a dynamic model of a Skeldar VTOL UAV. In the final phase, (d) a Skeldar UAVs running the optimal patrolling algorithm was used to provide protection for the ground allied teams in the evacuation mission.

For the case of the patrolling algorithm, all the adaptations towards more concrete levels of abstraction, i.e., into environment models having higher fidelity, were only a matter of slight adjustment of the algorithm implementation and posed no crucial problems. The most likely explanations behind the straightforward development process is the low computational complexity of the algorithm (the set of applicable strategies was precomputed) and a fundamental temporal flexibility in execution of such strategies.

2) Adversarial planning: modeling smart targets: The pursuit of an evading target was another subgoal of the evacuation tactical mission. The target was considered smart, i.e., a target, who actively monitors its surroundings and acts accordingly. Smart targets are aware of the fact that they are being tracked and actively try to avoid the tracking unit. Similarly, we consider trackers to be aware of the fact that the tracked targets are aware of their activities and try to act in the best response to the whole setup. Therefore, we need a formal game-theoretical model of a pursuit-evasion scenario with heterogeneous teams of agents and a resulting algorithm. The concrete goal of the algorithm was to control a team of assets (pursuers) attempting to detect, track and finally capture a number of smart targets (evaders) so that they act in the optimal way even against prospectively optimal evaders. Details of the used techniques can be found in [3], [12].

Analogically to the first mentioned algorithm, the analytical work and the reuse of state-of-the-art techniques led to an algorithm optimally controlling both the pursuing and evading agents involved in the game (see Figure 7). Firstly, (a) the theoretical analysis and theoretical guarantees of the algorithm were studied out using artificial and randomly generated graphs. After that, (b) the algorithm was experimented on the village street map, where pursuers and evaders move discretely in constant time steps. Finally, (c) the same algorithm was integrated into the evacuation mission scenario.

The adaptation of the algorithm among the different abstraction levels also turned out to be straightforward. This is due to the any-time property of the experimented algorithm.

3) Multi-agent re-planning and plan repair: Since tactical environments are typically highly dynamic, any planning algorithm has to consider failing actions. The more dynamic an environment is, the more actions of a plan fail. Classical-style planning is currently one of the most used techniques for automation of activities of intelligent agents. However, such plans are not robust in dynamic environments. The standard solution, in such cases, is to simply re-plan the agent's behavior from scratch and continue its actions according to the new plan. However, we have designed and adopted techniques preserving parts of the old plans – plan repair. The

(a) (b) (c)

Figure 8. Levels of abstraction used for design, verification and validation of the algorithm providing plan repairing ability for robotic assets: (a) synthetic environment used for the theoretical design, verification and validation of the plan repairing algorithms, (b) adaptation of the algorithm to a simplified dynamic model of Vidar VTOL UAVs providing support for the ground team (white lines represent the initial plan, yellow track represents a repaired plan), and (c) example of reconnaissance actions carried out by the Vidars in the integrated mission.

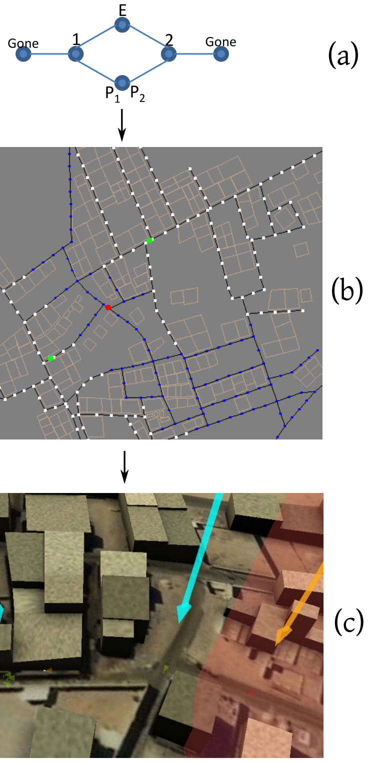


Figure 7. Levels of abstraction used for design, verification and validation of the algorithm solving the problem of a smart target pursuit: (a) example of a graph structure used for analysis of pathological instances of the pursuit-evasion game, (b) discrete movements based on a graph representing map of the streets, and (c) integrated scenario using interpolated movement for two blue-force troops (the blue arrows) dismounted from a team cargo truck pursuing high-value target adversary (the orange arrow).

main motivation is based on the assumption that the costs of communication in multi-agent teams is not negligible and therefore the algorithm should minimize it. Details about the employed plan repairing algorithm can be found in [3], [11].

The multi-agent plan repairing algorithms were based on classical plan repairing techniques and formally verified. The algorithms were used to control the Vidar VTOLs, which provide support for the ground team (see Figure 8). First tests and experiments were carried out in a synthetic grid-based environment (a) based on a classical state-of-the-art planning domain *crates-cranes*. Adaptation of the plan repairing algorithm to the VTOLs in the domain of tactical support (b) led to an introduction of a restricting condition on the depth of the search tree to limit the computational complexity of the search. Finally, (c) the constrained version of the algorithm was used in the integrated mission to provide visual support for the team. The plan-repairing mechanism is solving the problems caused by the unpredictable movement of the troops. The preconditions of the actions contained terms modeling the team has to be properly covered, and thus the initial plan has to be appropriately repaired during its movement.

During the adaptation process of the plan repairing algorithm, we have faced the problem with computational tractability and thus the maximum depth of the algorithm's search tree had to be limited. Such a change conditioned the soundness and completeness of the algorithm to only a short time horizons (equivalent to the length of the resulting plans).

4) Coordination and teamwork: Reactive planning is an alternative approach dealing with the dynamism of the environment, resulting plan failures and unexpected events. It allows a programmers to manually specify behaviors of agents in a rule-based language so that an agent's (robot's) action selection becomes efficient. The techniques we explored combined existing agent programming language Jazzyk[14] and a formalism for the specification of inter-agent coordination. These techniques were used in the evacuation mission to coordinate movement of the allied troops in formations and more importantly transitions among such formations. For more information about reactive multi-agent programming techniques used consult [14], [3].

The formation patterns were designed as short algorithms in an agent-programming language Jazzyk (see Figure 9). There were only two levels of abstraction used for verification and validation of the algorithms: (a) analytical design and synthetic testing of the patterns and (b) deployment of the algorithms in form of Jazzyk language programs and the related Jazzyk interpreter on the simulated troops.

The adaptation process involved mainly the implementation of glue code that specifies interactions between the different knowledge-base structures. Since Jazzyk language has been

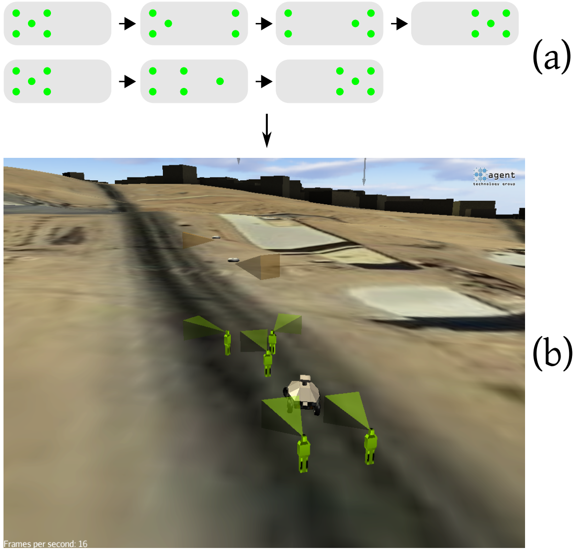


Figure 9. Levels of abstraction used for design and verification of multi-agent coordination algorithms: (a) an analytical design of an example coordination pattern, and (b) implementation of the pattern in the evacuation mission simulation.

design as highly elaboration tolerant, the patterns were easily tweaked to fit the rich simulated environment and the nature of its dynamics, e.g., recovering from collisions with obstacles and timing of the movements to synchronize with the other components of the system.

IV. Final Remarks

One can come up with various approaches to development of multi-agent applications for decentralized tactical missions. However, according to our experience such problems are typically so complex, that the first-shot approaches usually fail. We are providing a comprehensive description of a well-tried concept based on simulation-aided development methodology specifically focusing on the domain of tactical missions in dynamic environments.

We provide details to reproduce the process using any software solution available and suitable for the problem. Moreover, we give an overview of a software toolkit Alite and the way it could be customized towards a simulation system that is suitable for the simulation-aided development process.

Finally, we conclude the work with an example multi-agent application employing game-theoretic, plan repair and multi-agent coordination algorithms. The application demonstrates the use of such algorithms to control a robotic team that supports simulated troops in an evacuation tactical mission.

The most important direction for a future work is to provide well grounded processes along with software supporting automated, or at least semi-automated, design of the multi -level and -scope abstractions. Currently, all the levels have to be designed by hand and implemented on one-after-another basis, however, as shown in Section II-D, there are emerging patterns in the design of agent-to-environment interfaces and state descriptions. Exploitation of such patterns could allow more efficient development and faster advancement through the various abstraction levels towards the target systems.

Acknowledgements

This work was supported by U.S. Army Grant W911NF-10-1-0112 and by Czech Ministry of Education, Youth and Sports under Grant MSM6840770038.

The authors' organizations and research sponsors are authorized to reproduce and distribute reprints and on-line copies for their purposes notwithstanding any copyright annotation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of other parties.

References

[1] Jerry Banks, John Carson, Barry L. Nelson, and David Nicol. *Discrete-Event System Simulation (4th Edition)*. Prentice Hall, December 2004.

[2] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE wp, 2003.

[3] Branislav Bosansky, Michal Cap, Antonin Komenda, Viliam Lisy, Peter Novak, and Pechoucek Michal. Tactical AgentScout 2: Deliberative and reactive planning in adversarial environments – Final Report, April 2011.

[4] Branislav Bosansky, Viliam Lisy, Michal Jakob, and Michal Pechoucek. Computing time-dependent policies for patrolling games with mobile targets. In *Proceedings of The Tenth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, May 2011.

[5] Cougaar project website. http://www.cougaar.org/.

[6] M. Dickerson. Multi-agent simulation and NetLogo in the intro. comp. science curriculum. *J. Comput. Sci. Coll.*, 27:102–104, October 2011.

[7] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning about Knowledge*. MIT Press, 1995.

[8] Michal Jakob, Michal Pěchouček, Peter Novák, Michal Čáp, and Ondra Vaněk. Towards incremental development of human-agent-robot applications using mixed-reality testbeds. *IEEE Intelligent Systems, Special Issue on HART: Human-Agent-Robot Teamwork*, 2011. (accepted).

[9] Winnefeld A. James and Frank Kendall. Unmanned Systems Integrated Roadmap FY2011-2036, 2011.

[10] A. Komenda, J. Vokrinek, M. Pechoucek, G. Wickler, J. Dalton, and A. Tate. I-Globe: Distributed Planning and Coordination of Mixed-initiative Activities. In *Proceedings of Knowledge Systems for Coalition Operations (KSCO 2009)*, March-April 2009.

[11] Antonin Komenda and Peter Novak. Multi-agent plan repairing. In *Decision Making in Partially Observable, Uncertain Worlds: Exploring Insights from Multiple Communities, Proceedings of IJCAI 2011 Workshop*, pages 1–6. AAAI Press, 2011.

[12] Viliam Lisy, Michal Pechoucek, and Bosansky Branislav. Anytime algorithms for multi-agent visibility-based pursuit-evasion games. In *Proceedings of the Eleventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, 2012 (accepted).

[13] S. Luke, C. Cioffi-revilla, Panait L., and Sullivan K. Mason: A new multi-agent simulation toolkit. In *University of Michigan*, 2004.

[14] Peter Novák. *Jazzyk: A Programming Language for Hybrid Agents with Heterogeneous Knowledge Representations*, pages 72–87. Springer-Verlag, Berlin, Heidelberg, 2009.

[15] Michal Pěchouček, Michal Jakob, and Peter Novák. Towards simulation-aided design of multi-agent systems. In *Post-proceedings of the eighth international workshop on programming multi-agent systems, ProMAS 2010, LNAI, Vol. 6599*. Springer-Verlag, 2010. (in print).

[16] C. Siebra and A. Tate. I-Rescue: A Coalition Based System to Support Disaster Relief Operations. In *Proceedings of The Third International Association of Science and Technology for Development (IASTED) International Conference on Artificial Intelligence and Applications (AIA-2003)*, September 2003.

[17] David Šišlák, Milan Rollo, and Michal Pěchouček. A-Globe: Agent platform with inaccessibility and mobility support. In Matthias Klusch, Sascha Ossowski, Vipul Kashyap, and Rainer Unland, editors, *Cooperative Information Agents VIII*, volume 3191 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2004.

[18] J. Vokřínek, P. Novák, and A. Komenda. Ground Tactical Mission Support by Multi-agent Control of UAV Operations. volume 6867 of *Lecture Notes in Computer Science*, pages 225–234. Springer Berlin / Heidelberg, 2011.

[19] Jiří Vokřínek, Antonín Komenda, and Michal Pěchouček. Cooperative agent navigation in partially unknown urban environments. In *PCAR '10. Proceedings of the AAMAS-10 Workshops.*, pages 46–53, May 2010.