Linas Laibinis | Inna Pereverzeva | Elena Troubitsyna

Formal Reasoning about Resilient Goal-Oriented Multi-Agent Systems

Turku Centre *for* Computer Science

TUCS Technical Report
No 1133, April 2015

Formal Reasoning about Resilient Goal-Oriented Multi-Agent Systems

Linas Laibinis
Åbo Akademi University, Department of Computer Science
linas.laibinis@abo.fi
Inna Pereverzeva
Åbo Akademi University, Department of Computer Science
Turku Centre for Computer Science
inna.pereverzeva@abo.fi
Elena Troubitsyna
Åbo Akademi University, Department of Computer Science
elena.troubitsyna@abo.fi

TUCS Technical Report
No 1133, April 2015

Abstract

In this paper we present our formalisation of a resilient goal-oriented multi-agent system and its essential properties. The formalisation covers the notions of system goals and agents, various formal structures (functions and relations) defining different interrelationships between these notions, as well as constraints on the system dynamics allowing a multi-agent system to become more reconfigurable and thus resilient in order to achieve the system goals. The formalisation results in establishing connections between goals at different levels of abstraction, system architecture and agent responsibilities. The proposed formal systematisation of the involved concepts can be seen as generic guidelines for formal development of reconfigurable systems. Moreover, we demonstrate how such guidelines can be interpreted within the Event-B framework.

Keywords: Formal reasoning, Multi-agent system, Goal-oriented development, Resilience, Event-B

TUCS Laboratory
Embedded Systems Laboratory

1 Introduction

Resilience is an ability of a system to remain trustworthy despite changes [15]. It is an evolution of dependability concept that puts an emphasis on the ability of a system to adapt to different operating conditions. In this paper, we view adaptability as an ability of a system to reconfigure and continue to function in the presence of faults and other changes. Our aim is to propose a comprehensive theoretical study of relevant aspects of the system architecture and dynamic behaviour to facilitate formal development of reconfigurable distributed systems.

We consider distributed systems that are composed of asynchronously communicating heterogeneous components. The components interact with each other to execute functions required from the system. Moreover, to facilitate system resilience, the system components cooperatively perform fault tolerance activities as well as exchange information about their current status. The cooperative nature of the component behaviour makes it convenient to consider them as collaborating agents and the overall distributed system as a multi-agent system correspondingly.

Often research on multi-agent systems focuses on studying the emerging behaviour, i.e., it adopts a bottom-up approach that investigates whether agent interactions give rise to the desired behaviour or properties. In our work, we take an opposite approach: we aim at deriving the architectural and behavioural constraints to guarantee system resilience, i.e., ensure that the system, besides correct execution of its functions in the nominal conditions, can also reconfigure and remain operational in the presence of faults and other changes.

We rely on the goal-oriented paradigm because it provides us with a suitable conceptual basis for our reasoning. Goals are functional and non-functional[1] objectives that the system should achieve [34, 36]. High-level goals representing the overall system objectives can be decomposed into sub-goals. Decomposition facilitates unfolding of the layered system architecture and reasoning about system properties at different levels of abstraction. It also allows us to eventually derive constraints on the agent behaviour and ensure that their collaboration guarantees achieving the desired goals. The goal-oriented framework also provides us with an especially suitable basis for reasoning about reconfigurability. In particular, it allows us to define reconfigurability as an ability of agents to redistribute their responsibilities to ensure goal reachability.

Reasoning about reconfigurability within the goal-oriented multi-agent framework spans over a large set of inter-twined concepts addressing both system architecture and its dynamic behaviour. Therefore, there is a clear need for a formal systematic study of these complex interdependencies. This

[1]The non-functional aspect is not considered in the paper.

is the task that we tackle in this paper. Namely, we propose a systematic set-theoretic formalisation of the reconfigurability concept for multi-agent goal-oriented framework. The formalisation results in establishing connections between goals at different levels of abstraction, system architecture and agent responsibilities. The proposed formal systematisation of these concepts can be also seen as generic guidelines for formal development of reconfigurable systems. In this paper, we demonstrate how such guidelines can be interpreted within the Event-B framework [1].

The paper is structured as follows. Section 2 overviews the kind of systems and their properties we are interested in studying and briefly describes an illustrative example of such systems – a multi-robotic cleaning system. In Section 3 we gradually present our formalisation of a resilient goal-oriented multi-agent system and its reconfiguration mechanisms. Section 4 discusses how the formalised notions can be represented in a concrete formal framework – Event-B. Finally, we overview the related work and give some concluding remarks in Section 5.

2 Resilient Goal-Oriented Multi-Agent Systems

Resilience is an ability of a system to remain trustworthy despite changes [15]. To react on such changes, the system needs to reconfigure. The reconfiguration might be reactive or proactive. In the former case, reconfiguration is usually triggered by a component failure and the system should reconfigure to achieve fault tolerance, i.e., perform error recovery. In the latter case, the system might attempt to execute some of its services more efficiently, e.g., by deploying the available idle components. In both cases, the system components should collaborate to ensure system resilience.

In this paper, we study reconfigurability as an essential mechanism of achieving resilience of distributed systems. Since the collaborative aspect of the component behaviour is important for our study, we represent system components as agents and the overall system as a multi-agent system correspondingly.

Agents are autonomous software components that asynchronously communicate with each other. Each agent has a certain functionality that it provides. In this paper, we consider heterogeneous multi-agent systems, i.e., agents may have different functionalities. Moreover, some agents might play a role of supervisors of another agent or a group of agents. As a result of reconfiguration, an agent might receive additional responsibilities, i.e., it should become involved into an execution of tasks that were not allocated on it initially. We assume that agents are co-operative, i.e., they always accept new responsibilities. At the same time, the agents are unreliable, i.e., they

might fail and cease performing their functions. This might trigger system reconfiguration. As a result, the responsibilities of the failed agents can be re-allocated to the healthy ones. If an agent is healthy and idle, it can be deployed to perform the functions of failed agents or it might also become engaged into an execution of some other task, e.g., to improve the system performance and/or increase the likelihood of successful task completion.

While developing a multi-agent system, we should establish a link between system requirements and the agent behaviour. It is widely recognised that the goal-oriented development framework facilitates achieving this. The key concept of the framework is the notion of a goal – a functional or non-functional objective that a system should satisfy. Goals also constitute a convenient mechanism for structuring requirements via goal decomposition. In the decomposition process, the high-level system goals are iteratively decomposed into subgoals. Moreover, the low-level subgoals can be directly linked with the behaviour of agents, i.e., they can be used to derive requirements and constraints on the agent behaviour.

The goal-oriented framework provides us with a suitable basis for reasoning about reconfigurable multi-agent systems. It enables reasoning about the system behaviour at different levels of abstraction. At the same time, goal-decomposition process facilitates incremental unfolding of the system architecture. It also helps us to build a hierarchy of agents according to their responsibilities in achieving certain kind of goals. Moreover, the goal-oriented framework allows us to formulate reconfigurability as an ability of agents to redistribute their responsibilities to ensure goal reachability.

To summarise, in the rest of the paper we aim at studying the systems that have the following characteristics:

- There is a number of main (global) goals defined for the system. The goals can be decomposed into a subset of corresponding subgoals;
- The system consists of a number of agents – autonomic software components;
- The agents are organised hierarchically, i.e., one agent may be a supervisor of one or a group of other agents;
- The agents interact with each other in order to achieve the system goals;
- In general, agent interactions can vary from simple information exchanges to requests for specific actions or services to be performed;
- The system agents are unreliable components that might fail during system execution;

- In the case of agents failures, the system should, if possible, dynamically reconfigure itself to achieve the overall system goal;

- The system can also reconfigure to achieve some of its goals more efficiently by means of, e.g., deploying idle agents.

Next we describe our running example – a *multi-robotic cleaning system* – that can be considered as an illustrative instance of the systems whose properties we described above.

A Multi-Robotic Cleaning System. The *main goal* of the multi-robotic system is to get a certain area cleaned by the means of involved system agents. There are two *types of agents* that are responsible for achieving the goal. The first type is base stations – the stationary devices that coordinate cleaning activities by assigning cleaning tasks to the second type of agents – robots. The robots are autonomous electro-mechanical devices that can move, clean, as well as communicate with the base stations. Both base stations and robots are unreliable, i.e., they can fail at any moment. In the case of these failures, the system should, if possible, *dynamically reconfigure* itself to achieve the overall system goal.

The whole territory to clean is divided into several zones, which are further divided into a number of sectors. To clean the territory, every its zone has to be cleaned. In its turn, to clean a zone, every its sector has to be cleaned. Each zone has the associated base station that *coordinates* the cleaning activities within the zone. In general, the coordination activities of one base station may span over several zones. Moreover, each base station *supervises* a number of robots attached to it by *assigning* cleaning tasks to them.

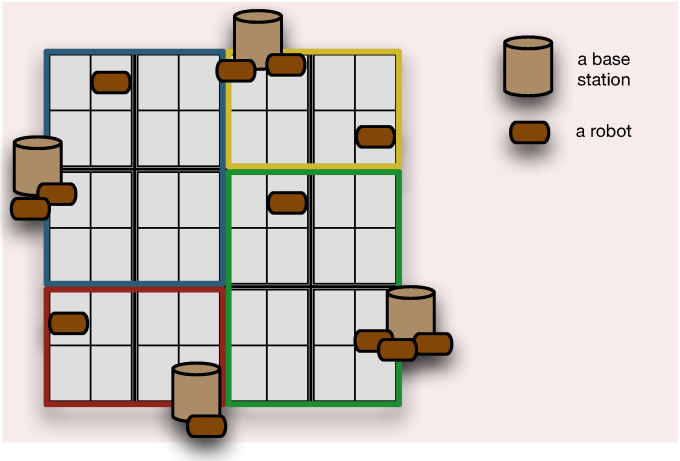

Figure 1: Multi-Robotic Cleaning System

A base station might assign a robot a specific sector to clean. Upon receiving a cleaning assignment, the robot autonomously moves to this sector and performs cleaning. After successfully completing its mission, the robot returns back to the base station to receive a new assignment. The base station keeps track of the cleaned and non-cleaned sectors. Moreover, the base stations periodically exchange the information about their cleaned sectors.

While performing a given task, a robot might fail at any moment. In that case, the base station may assign another active robot to perform the failed task. A base station might fail as well. In that case, the healthy base stations should redistribute the responsibility over the zones as well as the control over the associated robots of the failed base station.

As we can see, such a multi-robotic system exhibits the general characteristics and properties that we described above. We are going to use this system as the running example for the rest of this paper.

3 Formalisation of a Resilient Goal-Oriented MAS

In this section we present our formalisation of a resilient goal-oriented system and its essential properties. The formalisation will cover the notions of system goals and agents, various formal structures (functions and relations) defining different interrelationships between these notions, as well as constraints on the system dynamics allowing a multi-agent system to become more reconfigurable and thus resilient in order to achieve the system goals. The formalisation summarises our experience in formal modelling and verification of resilient goal-oriented multi-agent systems [22, 23, 24, 32, 13].

Notational conventions. In addition to the standard set-theoretical notation we are going rely on (e.g., $\in, \subseteq, \cap, \cup, \varnothing$, etc), the operator $\setminus$ is used to denote set subtraction. $T_1 \times T_2$ is a cartesian product of two types (sets) T_1 and T_2. The notation $\mathcal{P}(T)$ stands for the powerset (set of all subsets) type over elements of the type T, while $T_1 \leftrightarrow T_2$ denotes a relation between elements of two types (sets), i.e.,

$$R : T_1 \leftrightarrow T_2 \;\Leftrightarrow\; R : \mathcal{P}(T_1 \times T_2).$$

Moreover, **dom** and **ran** are respectively the relation domain and range operators, while $R_1; R_2$ stands for relational composition of two relations R_1 and R_2. Id represents the identity relation, while R^* denotes the reflexive transitive closure of a relation R, i.e.,

$$R^* \;=\; Id \cup R \cup R;R \cup R;R;R \cup \ldots .$$

We will also use the transitive closure of a relation R, denoted as R^+ and defined as $R^+ = R^* \backslash Id$, or

$$R^+ = R \cup R;R \cup R;R;R \cup \ldots .$$

We will treat functions as a special kind of relations and relations as a special kind of sets (i.e., sets of pairs, triples, etc). The notation $(e_1 \mapsto e_2) \in R$ will be used to check that two elements are related by the binary relation or function R. Similarly, $(e_1 \mapsto e_2 \mapsto e_3) \in R$ will be used for checking membership in a ternary relation.

3.1 A Goal-oriented State Transition System

We are going to build our formalisation of a goal-oriented multi-agent system by gradually extending the standard definition of a state transition system, typically defined as a triple $(\Sigma, \mathit{Init}, \mathit{Trans})$, where Σ represents all the system states, *Init* stands for its possible initial states, and *Trans* defines all the allowed transitions between system states. We start by introducing the notion of system goals that such a state transition system should try to achieve.

Definition 1 Goal-oriented state transition system (GSTS). *A GSTS system is a tuple $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap})$, where $\mathcal{G}$ is a set of all possible system goals, Σ is the system state space, Init is a set of initial system states, Trans is a next-state relation of a GSTS system, and GMap is a function mapping a system goal to a subset of system states, such that*

(1.1) $\mathit{Init} : \mathcal{P}(\Sigma)$,
(1.2) $\mathit{Trans} : \Sigma \leftrightarrow \Sigma$,
(1.3) $\mathit{GMap} : \mathcal{G} \to \mathcal{P}(\Sigma)$,
(1.4) $\mathit{Init} \neq \varnothing$,
(1.5) $\mathit{Init} \subseteq \mathbf{dom}(\mathit{Trans})$,
(1.6) $\forall g : \mathcal{G}.\ \mathit{GMap}(g) \neq \varnothing$,
(1.7) $\forall g : \mathcal{G}.\ \exists \sigma, \sigma' : \Sigma.\ \sigma \in \mathit{Init} \wedge (\sigma \mapsto \sigma') \in \mathit{Trans}^* \wedge \sigma' \in \mathit{GMap}(g)$.

The required properties (1.1), (1.2), (1.4), and (1.5) are inherited from the standard definition of a state transition system. The two new elements of a GSTS introduce the abstract notion of system goals (as the type $\mathcal{G}$) and relate these goals (via the function *GMap*) with specific system states where these goals are considered to be achieved. Essentially, the function *GMap* assigns semantics to any goal from $\mathcal{G}$ by associating it with a non-empty set of states (a predicate) of Σ, as stated in (1.3) and (1.6). Finally, the last (1.7)

property of a GSTS requires that all the system goals must be achievable after system initialisation, i.e., they should be true either initially or after a number of system transitions defined by *Trans*.

Let us recall our running example – the multi-robotic cleaning system described in Section 2. The set $\mathcal{G}$ of this system contains two kinds of goals: "The zone j must be cleaned", for any $j \in 1..NumberOfZones$, and "The sector i of the zone j must be cleaned", for any $i \in 1..NumberOfSectors$ and $j \in 1..NumberOfZones$.

What would be a possible definition of *GMap* for this system? Let us consider the goal $g =$"The sector k of the zone l must be cleaned", for some fixed $k \in 1..NumberOfSectors$ and $l \in 1..NumberOfZones$. The mapping $GMap(g)$ then may be defined as, e.g.,

$$GMap(g) \;=\; \{\sigma \mid \; (SectorCleaned(\sigma))[l,k] \;=\; TRUE\},$$

where *SectorCleaned* is a state variable (binary array) storing information about the cleaned sectors. The type of such a variable is

$$\Sigma \to (1..NumberOfZones \times 1..NumberOfSectors \to BOOL).$$

An important dynamic property of a GSTS system is *stability* with respect to its goals, i.e., the system ability to retain the goals that have been already achieved.

Definition 2 Stable GSTS. *A GSTS system* $(\mathcal{G}, \Sigma, Init, Trans, GMap)$ *is called* stable *if*

$$\textit{(2.1)}\;\; \forall \sigma, \sigma' : \Sigma, g : \mathcal{G}.\; (\sigma \mapsto \sigma') \in Trans \;\wedge\; \sigma \in GMap(g) \;\Rightarrow\; \sigma' \in GMap(g)$$

The system stability is a very desirable system property to have (especially for formal verification), however, it is also quite strong constraint on the system behaviour. For our example of the cleaning system, such an assumption would mean that all the cleaning goals are achievable in short duration, i.e., none of the cleaned sectors or zones gets "dirty" again before system termination.

So far, we considered system goals as members of a given set, which can be pursued and accomplished completely independently. Often, we can talk about some structure introducing inter-relationships between the system goals, e.g., distinguishing particular goals and their subgoals. This also implies that their semantic definitions (i.e., *GMap* functions) should be interrelated too. We can define these interrelationships by introducing two new structures – *G_graph* and *SGMap*.

Definition 3 Structured GSTS. *A GSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap})$ *is called* structured *if exist a relation on goals G_graph and a function SGMap, such that*

$$\begin{array}{ll} (3.1) & \mathit{G\_graph} : \mathcal{G} \leftrightarrow \mathcal{G}, \\ (3.2) & \mathit{SGMap} : \mathcal{G} \to \mathcal{P}(\Sigma), \\ (3.3) & \forall g : \mathcal{G}.\ (g \mapsto g) \notin \mathit{G\_graph}^+, \\ (3.4) & \forall g : \mathcal{G}.\ \mathit{GMap}(g) \subseteq \mathit{SGMap}(g), \\ (3.5) & \forall g, g' : \mathcal{G}.\ g' \in \mathit{Subgoals}(g) \Rightarrow \mathit{SGMap}(g) \cap \mathit{GMap}(g') \neq \varnothing, \end{array}$$

where $\mathit{Subgoals}(g) = \{g' : \mathcal{G} \mid (g \mapsto g') \in \mathit{G\_graph}\}$.

Moreover, the following is true

$$(3.6) \quad \forall \sigma : \Sigma, g : \mathcal{G}.\ \sigma \in \mathit{SGMap}(g) \wedge \sigma \notin \mathit{GMap}(g) \Rightarrow \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans} \wedge \sigma' \in \mathit{GMap}(g).$$

Mathematically, *G_graph* stands for an acyclic graph on goals. It describes relationships between different goals, e.g., how a particular goal can be split into its subgoals and so on. The property (3.3) states that a goal cannot be a subgoal of itself, i.e., the graph does not contains loops. The properties (3.4) - (3.6) give an alternative definition of the states associated with a particular goal, given as a function *SGMap*, in connection to the corresponding states of its subgoals. Specifically, (3.4) states that achieving g according to *GMap* implies that the goal g was achieved according to *SGMap* as well, while (3.5) requires that achieving any of subgoals must contribute to that of the parent goal. Intuitively, $\mathit{SGMap}(g)$ stands for the necessary precondition for achieving g, relating it with an arbitrary expression on the subgoals of g.

Finally, the last property (3.6) requires that, once the associated expression on subgoals *SGMap* is satisfied, the system always has an opportunity (a respective state transition) to complete the parent goal g, i.e., reach a state from *GMap(g)*. We deliberately allow such a "gap" in terms of an extra transition between achieving the main goal and that of its subgoals in the system dynamics because, as we will see later, achieving different goals can be responsibility of different system agents.

Let us go back to our running example. Since any zone is considered cleaned only after all its sectors are cleaned, the relation *G_graph* can be simply defined as

$$\{\text{"The zone } j \text{ must be cleaned"} \mapsto \text{"The sector } i \text{ of the zone } j \text{ must be cleaned"} \mid i \in 1..\mathit{NumberOfSectors} \wedge j \in 1..\mathit{NumberOfZones}\}.$$

Moreover, for the goal $g =$ "The zone l must be cleaned", for some fixed $l \in 1..NumberOfZones$, and its subgoals $Subgoals(g) = \{$"The sector k of the zone l must be cleaned" $\mid k \in 1..NumberOfSectors\}$, the mapping $SGMap(g)$ can be defined as

$$SGMap(g) \;=\; \{\sigma \mid \forall k \in 1..NumberOfSectors.\, (SectorCleaned(\sigma))[l,k] = TRUE\},$$

where *SectorCleaned* is a state variable described above.

Having the goal structure defined, we can easily distinguish the top goals of a structured GSTS. These are the goals that do not participate as subgoals for any other goal.

Definition 4 Top goals. *For a structured GSTS and its relation on goals G_graph, the system top goals are defined as*

$$(4.1) \qquad TopG \;=\; \mathbf{dom}(G\_graph) \backslash \mathbf{ran}(G\_graph).$$

Since *G_graph* is acyclic, the set *TopG* is always non-empty.

In particular, the top goals are especially important when we consider terminating GSTSs. The system termination can be easily formally defined by analysing their next state relation *Trans* as follows.

Definition 5 Terminating GSTS. *A GSTS is terminating if*

$$(5.1) \qquad \mathbf{ran}(Trans) \backslash \mathbf{dom}(Trans) \;\neq\; \varnothing.$$

When such a system terminates, we usually expect a certain property to be true on its top goals. A concrete choice depends of course on the considered system. Two obvious solutions are formalised below. The first one requires that the system in its terminating states achieves all its goals:

$$\forall \sigma : \Sigma, g : \mathcal{G}.\; \sigma \in \mathbf{ran}(Trans) \backslash \mathbf{dom}(Trans) \;\wedge\; g \in TopG \;\Rightarrow\; \sigma \in GMap(g).$$

Alternatively, we can require that at least one top goal is achieved:

$$\forall \sigma : \Sigma.\; \sigma \in \mathbf{ran}(Trans) \backslash \mathbf{dom}(Trans) \;\Rightarrow\; \exists g : \mathcal{G}.\; g \in TopG \wedge \sigma \in GMap(g).$$

For our running example, the top goals are obviously those that are related with zone cleaning. In the terminating states of the system, the cleaning system is required to achieve all its goals (a property of the first kind). Alternatively, the system could have a failsafe mechanism installed, which must be activated when completion of the cleaning (for whatever reason) becomes impossible. In this case, a property of the second kind can be enforced, requiring that either all the zones are cleaned or the failsafe procedure is successfully finished.

3.2 Introducing Agents

Now we extend the definition of a goal-oriented state transition system presented in the previous section by introducing agents that can carry out tasks leading to achieving the system goals.

Definition 6 Multi-agent goal-oriented state transition system (MAGSTS). *A MAGSTS system is a tuple* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *such that* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap})$ *is a GSTS,* $\mathcal{A}$ *is a set of all system agents, and* Active *is a function returning a subset of active agents in a particular system state, where*

$$\mathit{Active} : \Sigma \to \mathcal{P}(\mathcal{A}). \tag{6.1}$$

The definition introduces a type (set) $\mathcal{A}$ for all possible system agents and also associates a subset of active agents in the current system state via the function *Active*. Our interpretation of "active" agents is that only active agents can carry out the tasks in order to achieve the system goals. Inactive agents are either those are not currently present in the system or those that are failed and thus incapable to carry out any tasks.

If a multi-agent systems allows the agents to become active or inactive (e.g., failed) at any moment, we call such a system open. Formally, we define it as follows.

Definition 7 Open MAGSTS. *A MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *is open if the following properties hold:*

$$\begin{aligned} (7.1) \quad & \forall \sigma : \Sigma,\ a : \mathcal{A}.\ \sigma \in \mathbf{dom}(\mathit{Trans}) \ \wedge\ a \in \mathit{Active}(\sigma) \ \Rightarrow \\ & \exists \sigma'.\ (\sigma \mapsto \sigma') \in \mathit{Trans} \wedge \mathit{Active}(\sigma') = \mathit{Active}(\sigma) \backslash \{a\} \end{aligned}$$

and

$$\begin{aligned} (7.2) \quad & \forall \sigma : \Sigma,\ a : \mathcal{A}.\ \sigma \in \mathbf{dom}(\mathit{Trans}) \ \wedge\ a \notin \mathit{Active}(\sigma) \ \Rightarrow \\ & \exists \sigma'.\ (\sigma \mapsto \sigma') \in \mathit{Trans} \wedge \mathit{Active}(\sigma') = \mathit{Active}(\sigma) \cup \{a\}. \end{aligned}$$

In our running example, the set $\mathcal{A}$ include all the system base stations and robots, active as well as inactive ones. Both base stations and robots can fail at any moment, thus becoming inactive. If we require the described cleaning system open, this would mean that some recovery mechanism must be in place, allowing any failed base station or robot to be reintroduced into the system as an active agent.

Since the set $\mathcal{A}$ contains all possible system agents, some of them may have very different functionalities (abilities). In order to associate certain classes of agents with specific types of system goals they are able to accomplish, we first introduce classifications of system agents and goals and then define relationships between the introduced classes.

Definition 8 Typed MAGSTS. *A structured MAGSTS system* $(\mathcal{G}, \Sigma, Init, Trans, GMap, \mathcal{A}, Active)$ *is typed if there exist the functions atype and gtype, such that*

$$\begin{aligned}
&(8.1) \quad atype : \mathcal{A} \to AType,\\
&(8.2) \quad gtype : \mathcal{G} \to GType,\\
&(8.3) \quad \forall at : AType.\ \exists a : \mathcal{A}.\ atype(a) = at,\\
&(8.4) \quad \forall gt : GType.\ \exists g : \mathcal{G}.\ gtype(g) = gt,\\
&(8.5) \quad \forall g_1, g_2 : \mathcal{G},\ gt : GType.\ gtype(g_1) = gt \ \wedge\ gtype(g_2) = gt \ \wedge\ g_1 \neq g_2 \ \Rightarrow\\
&\qquad\qquad GMap(g_1) \cap GMap(g_2) = \varnothing,
\end{aligned}$$

where AType and GType are abstract types containing all possible agent and goal types respectively.

In (8.1) and (8.2), the functions *atype* and *gtype* associate each agent and goal with their respective type. Both agent and goal types are nonempty in the sense that they must have at least one agent or goal associated with them (properties (8.3) and (8.4)). The last property (8.5) is introduced to ensure that distinct goals of the same goal type can be achieved independently, i.e., can be assigned to different agents to accomplish them in parallel. However, before giving such assignments to agents, we have to be sure that they are able to accomplish the assigned tasks. To formalise this, we introduce a special relation to represent interrelationships between different agent and goal types.

Definition 9 Relationship between agent and goal types *We say that agent and goal types are related if there exists a relation AG_Rel, such as*

$$\begin{aligned}
&(9.1) \quad AG_Rel : AType \leftrightarrow GType,\\
&(9.2) \quad \mathbf{dom}(AG_Rel) \ =\ AType, \ and\\
&(9.3) \quad \mathbf{ran}(AG_Rel) \ =\ GType.
\end{aligned}$$

For convenience, the relation *AG_Rel* can be represented as a pair of functions A_goals, $A\_goals : AType \to \mathcal{P}(GType)$, and G_agents, $G\_agents : GType \to \mathcal{P}(AType)$, such that

$$\forall at : AType, gt : GType.\ gt \in A\_goals(at) \ \Leftrightarrow\ (at \mapsto gt) \in AG\_Rel$$

and

$$\forall gt : GType, at : AType.\ at \in G\_agents(gt) \ \Leftrightarrow\ (at \mapsto gt) \in AG\_Rel.$$

From this definition, we immediately get that

$$\forall at : AType, gt : GType.\ gt \in A\_goals(at) \ \Leftrightarrow\ at \in G\_agents(gt)$$

In our running example, *AType* can be easily defined as the set $\{BaseStations, Robots\}$, while *GType* is simply $\{CleaningZones, CleaningSectors\}$. The relation *AG_Rel* then interconnects the introduced agent and goal types as follows:

$$\{BaseStations \mapsto CleaningZones, Robots \mapsto CleaningSectors\}.$$

Knowing the interrelationships between the agent and goal types allows us to check in a straightforward way whether a concrete agent is able to accomplish a specific goal.

Definition 10 Agent ability *We say that an agent $a : \mathcal{A}$ is able to accomplish a goal $g : \mathcal{G}$ if*

$$(10.1) \qquad atype(a) \in G\_agents(gtype(g))$$

or, equivalently,

$$(10.2) \qquad gtype(g) \in A\_goals(atype(a)).$$

Often, the hierarchical structure between goals and subgoals, formalised by *G_graph*, is reflected on the goal types as well.

Definition 11 Hierarchy of goal types *We say a structured MAGSTS system supports a hierarchy of goal types if there is exists a relation GT_graph, such that*

$$(11.1) \qquad GT\_graph : GType \leftrightarrow GType,$$

$$(11.2) \qquad \forall gt \in GType.\ (gt \mapsto gt) \notin GT\_graph^{+},$$

$$(11.3) \qquad \forall g_1, g_2 : \mathcal{G}.\ (gtype(g_1) \mapsto gtype(g_2)) \in GT\_graph \ \Rightarrow$$
$$gtype(g_1) \neq gtype(g_2) \ \wedge\ (g_1 \mapsto g_2) \in G\_graph^{+}.$$

For our running example, *GT_graph* is simply a singleton set $\{CleaningZones \mapsto CleaningSectors\}$.

The hierarchical structures between goals and subgoals introduced above define the existing dependencies between the goals and thus imply the manner their achievement can be coordinated among the involved agents. Moreover, the formalised connection between the agent and goal types clarifies which agents can be given the tasks related with specific system goals.

3.3 Agent Subordination and Supervision

Having agent types and hierarchy of goal types defined makes it possible to introduce a subordination structure between agent types.

Definition 12 Subordinated MAGSTS. *A MAGSTS system* $(\mathcal{G}, \Sigma, Init, Trans, GMap, \mathcal{A}, Active))$ *is called* subordinated *if it is typed, supports a hierarchy of goal types, and exists a relation on agent types A_Sub, such that*

$$\begin{aligned}
&(12.1) \quad A_Sub : AType \leftrightarrow AType,\\
&(12.2) \quad \mathbf{dom}(A_Sub) \cup \mathbf{ran}(A_Sub) = AType,\\
&(12.3) \quad \forall at \in AType.\ (at \mapsto at) \notin A_Sub^{+}.
\end{aligned}$$

Moreover, for each $at_1, at_2 : AType$*, such that* $(at_1 \mapsto at_2) \in A\_Sub$*, the following property must hold*

$$\begin{aligned}
(12.4) \quad &\exists gt_1, gt_2 : GType.\\
&(gt_1 \mapsto gt_2) \in GT_graph \ \wedge \ gt_1 \in A_goals(at_1) \wedge gt_2 \in A_goals(at_2).
\end{aligned}$$

According to the definition (properties (12.1)–(12.3)), *A_Sub* is acyclic graph covering all system agent types. The last property (12.4) states the required connection between two hierarchical structures: the goal type structure *GT_graph* and the agent subordination structure *A_Sub*. Namely, for each pair of subordinated agent types, exists (at least one) pair of the related goal types such that goals of the parent goal type can be handled by agents of the "master" agent type, while goals of the subgoal type can be handled by agents of the subordinate agent type.

It is obvious that, for our running example, the only possible way to define *A_Sub* is as a singleton set $\{BaseStations \mapsto Robots\}$. The base stations are responsible for zone cleaning, which can be decomposed into cleaning of the constituent sectors by robots. Since base stations are responsible for a higher level goal (i.e., are aware of a "bigger picture"), it is natural to appoint them as supervisors with respect to robots.

If a system is centralised one, even members of the top agent type may be needed to be supervised. In that case, the agent type hierarchy can be artificially extended with the top element *SystemType*, which has a single agent *System* as its member.

Similarly as for *AG_Rel*, the relation *A_Sub* can be represented as a pair of functions *AS_goals*, $AS\_goals : AType \to \mathcal{P}(GType)$, and *GS_agents*, $GS\_agents : GType \to \mathcal{P}(AType)$, such that

$$\begin{aligned}
&\forall at : AType, gt : GType.\ gt \in AS_goals(at) \ \Leftrightarrow\\
&\quad \exists gt' : GType.\ (gt' \mapsto gt) \in GT_graph \ \wedge \ gt' \in A_Goals(at)
\end{aligned}$$

and

$$\forall gt : GType, at : AType.\ at \in GS\_agents(gt) \ \Leftrightarrow \\ \exists gt' : GType.\ (gt' \mapsto gt) \in GT\_graph \ \wedge \ gt' \in A\_Goals(at).$$

From this definition, we immediately get that

$$\forall at : AType, gt : GType.\ gt \in AS\_goals(at) \ \Leftrightarrow \ at \in GS\_agents(gt)$$

The above definitions allow us to check in a straightforward way whether a concrete agent is able to supervise accomplishing a specific goal.

Definition 13 Agent supervision *We say that an agent* $a : \mathcal{A}$ *is able to supervise a goal* $g : \mathcal{G}$ *if*

$$(13.1) \qquad atype(a) \in GS\_agents(gtype(g))$$

or, equivalently,

$$(13.2) \qquad gtype(g) \in AS\_goals(atype(a)).$$

The notions about agents introduced so far (agent types, subordination, ability to accomplish or supervise a particular goal) define required static properties of a multi-agent goal-oriented system. The only exception is a function *Active*, which returns a set of active agents in a particular system state. Since agents can change their active/inactive status during system execution, the function expresses a dynamic system characteristic.

In subordinated MAGSTSs, a part of system agents supervise activities of other agents. Moreover, they can give concrete goal assignments to subordinate agents, which, in turn, should "report" to their supervisors once the assigned goal has been accomplished. The unreached system goals can be also dynamically partitioned among the supervisor agents.

This allows us to introduce a few additional dynamic system characteristics. Namely, in a specific dynamic system state, a particular agent can be "attached" to another agent, which serves as its supervisor. A specific, yet unreached goal can be put under responsibility of a particular supervisor agent. Finally, a specific goal can be "assigned" by a supervisor to one of its subordinate agents.

Let us now to define these dynamic notions formally.

Definition 14 Agent attachment *A MAGSTS system* $(\mathcal{G}, \Sigma, Init, Trans, GMap, \mathcal{A}, Active)$ *supports agent attachment if there is a dynamic attribute*

(function) Attached, such that

$$\textit{(14.1)} \quad \textit{Attached} : \Sigma \to \mathcal{P}(\mathcal{A} \times \mathcal{A}),$$

$$\begin{aligned}\textit{(14.2)} \quad & \forall \sigma : \Sigma,\ a_1, a_2 : \mathcal{A}.\ (a_1 \mapsto a_2) \in \textit{Attached}(\sigma)\ \Rightarrow \\ & a_1 \in \textit{Active}(\sigma)\ \wedge\ a_2 \in \textit{Active}(\sigma)\ \wedge\ \textit{atype}(a_1) \mapsto \textit{atype}(a_1) \in \textit{A\_Sub}\ \wedge \\ & \neg(\exists a_3 : \mathcal{A}.\ a_3 \neq a_1\ \wedge\ (a_3 \mapsto a_2)\ \in\ \textit{Attached}(\sigma)).\end{aligned}$$

Moreover, the following property is true

$$\begin{aligned}\textit{(14.3)} \quad & \forall \sigma : \Sigma,\ a_1, a_2 : \mathcal{A}.\ a_1 \in \textit{Active}(\sigma)\ \wedge\ a_2 \in \textit{Active}(\sigma)\ \wedge \\ & \textit{atype}(a_1) \mapsto \textit{atype}(a_2) \in \textit{A\_Sub}\ \wedge\ \neg(\exists a_1' : \mathcal{A}.\ (a_1' \mapsto a_2)\ \in\ \textit{Attached}(\sigma)) \\ & \Rightarrow \\ & \qquad \exists(\sigma' : \Sigma).\ (\sigma \mapsto \sigma') \in \textit{Trans}\ \wedge\ (a_1 \mapsto a_2) \in \textit{Attached}(\sigma').\end{aligned}$$

Therefore, for any agents a_1, a_2 and system state σ, the expression $(a_1 \mapsto a_2) \in \textit{Attached}(\sigma)$ implies that (i) both agents are active in σ, (ii) the agent type of a_2 is subordinate to that of a_1, and (iii) the agent a_2 is not currently attached to any other supervisor agent (property (14.2)).

Moreover, a MAGST system supports agent attachment if, at any point where the conditions for agent attachment are satisfied, the system has an opportunity (but not an obligation) to do such an action (property (14.3)).

In our running example, any active and yet unattached robot can be attached to any active base station. It can also change its supervisor base station to a different active one.

Definition 15 Goal responsibility *A MAGSTS system* $(\mathcal{G}, \Sigma, \textit{Init}, \textit{Trans}, \textit{GMap}, \mathcal{A}, \textit{Active})$ *supports goal responsibility if there is a dynamic attribute (function) Responsible, such that*

$$\textit{(15.1)} \quad \textit{Responsible} : \Sigma \to \mathcal{P}(\mathcal{G} \times \mathcal{A}),$$

$$\begin{aligned}\textit{(15.2)} \quad & \forall \sigma : \Sigma,\ g : \mathcal{G},\ a : \mathcal{A}.\ (g \mapsto a) \in \textit{Responsible}(\sigma)\ \Rightarrow \\ & \quad a \in \textit{Active}(\sigma)\ \wedge\ \textit{gtype}(g) \in \textit{AS\_goals}(\textit{atype}(a))\ \wedge \\ & \quad \neg(\exists a' : \mathcal{A}.\ a' \neq a\ \wedge\ (g \mapsto a')\ \in\ \textit{Responsible}(\sigma)).\end{aligned}$$

Moreover, the following property is true

$$\begin{aligned}\textit{(15.3)} \quad & \forall \sigma : \Sigma,\ g : \mathcal{G},\ a : \mathcal{A}.\ a \in \textit{Active}(\sigma)\ \wedge\ \textit{gtype}(g) \in \textit{AS\_goals}(\textit{atype}(a))\ \wedge \\ & \neg(\exists a' : \mathcal{A}.\ (g \mapsto a')\ \in\ \textit{Responsible}(\sigma)) \\ & \Rightarrow \\ & \qquad \exists(\sigma' : \Sigma).\ (\sigma \mapsto \sigma') \in \textit{Trans}\ \wedge\ (g \mapsto a) \in \textit{Responsible}(\sigma').\end{aligned}$$

Therefore, for any goal g, agent a and system state σ, the expression $(g \mapsto a) \in \mathit{Responsible}(\sigma)$ implies that (i) the agent a is active in the state σ, (ii) the agent type allows it to supervise g, and (iii) the goal g is not currently under responsibility of any other supervisor agent (property (15.2)).

Moreover, a MAGST system supports goal responsibility if, at any point where the conditions for an agent taking responsibility for some goal are satisfied, the system has an opportunity (but not an obligation) to do this action (property (15.3)).

In our running example, any active base station can take responsibility over a zone which is not yet responsibility of any other base station. Zone responsibility can also "migrate" from one base station to another as a part of the system reconfiguration.

Definition 16 Goal assignment *A MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *supports goal assignment if there is a dynamic attribute (function) Assigned, such that*

(16.1) $\quad \mathit{Assigned} : \Sigma \to \mathcal{P}(\mathcal{G} \times \mathcal{A} \times \mathcal{A}),$

(16.2)
$$\begin{aligned}
&\forall \sigma : \Sigma,\ g : \mathcal{G},\ a_1, a_2 : \mathcal{A}.\ (g \mapsto a_1 \mapsto a_2) \in \mathit{Assigned}(\sigma)\ \Rightarrow \\
&(a_1 \mapsto a_2) \in \mathit{Attached}(\sigma)\ \wedge\ (g \mapsto a_1) \in \mathit{Responsible}(\sigma)\ \wedge \\
&\mathit{gtype}(g) \in \mathit{A_goals}(\mathit{atype}(a_2))\ \wedge \\
&\neg(\exists g' : \mathcal{G},\ a' : \mathcal{A}.\ g' \neq g\ \wedge\ a'_1 \neq a_1\ \wedge\ (g' \mapsto a'_1 \mapsto a_2)\ \in\ \mathit{Assigned}(\sigma))\ \wedge \\
&\neg(\exists a'_1, a'_2 : \mathcal{A}.\ a'_1 \neq a_1\ \wedge\ a'_2 \neq a_2\ \wedge\ (g \mapsto a'_1 \mapsto a'_2)\ \in\ \mathit{Assigned}(\sigma))\ \wedge \\
&\sigma \notin \mathit{GMap}(g)\ \wedge\ \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ \sigma' \in \mathit{GMap}(g).
\end{aligned}$$

Moreover, the following property is true

(16.3)
$$\begin{aligned}
&\forall \sigma : \Sigma,\ g : \mathcal{G},\ a : \mathcal{A}.\ (a_1 \mapsto a_2) \in \mathit{Attached}(\sigma)\ \wedge \\
&\mathit{gtype}(g) \in \mathit{A_goals}(\mathit{atype}(a_2))\ \wedge\ (g \mapsto a_1) \in \mathit{Responsible}(\sigma)\ \wedge \\
&\neg(\exists g' : \mathcal{G},\ a' : \mathcal{A}.\ g' \neq g\ \wedge\ a'_1 \neq a_1\ \wedge\ (g' \mapsto a'_1 \mapsto a_2)\ \in\ \mathit{Assigned}(\sigma))\ \wedge \\
&\neg(\exists a'_1, a'_2 : \mathcal{A}.\ (g \mapsto a'_1 \mapsto a'_2)\ \in\ \mathit{Assigned}(\sigma)) \\
&\Rightarrow \\
&\exists (\sigma' : \Sigma).\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (g \mapsto a_1 \mapsto a_2) \in \mathit{Assigned}(\sigma'),
\end{aligned}$$

Therefore, for any agents a_1, a_2, a goal g and system state σ, the expression $(g \mapsto a_1 \mapsto a_2) \in \mathit{Assigned}(\sigma)$ implies that (i) a_2 is attached to a_1 in the state σ, (ii) a_1 is responsible for achieving the goal g in the state σ, (iii) a_2 is able to accomplish any goal of the type $\mathit{gtype}(g)$, (iv) the agent a_2 is not assigned to any other goal, (v) the goal g is not assigned to any other agent, (vi) the goal g is not yet completed, and (vii) once the goal g is

assigned, it can be completed at any moment (property (16.2)). The last two properties allow us to associate the goal reachability with goal assignment, and by transitivity, goal responsibility and agent attachment mechanisms.

Moreover, a MAGST system supports goal assignment if, at any point where the conditions for goal assignment are satisfied, the system has an opportunity (but not obligation) to do this action (property (16.3)).

In our running example, any attached and active robot without the current cleaning assignment (e.g., just after finishing the previous one) can be given a new cleaning assignment by its supervisor base station.

3.4 System Reconfiguration and Goal Reachability in the Presence of Agent Failures

Even though the above definitions require the existence of system transitions for the agents and goals that are "free", i.e., have not been attached or assigned, they implicitly cover two more kinds of system transitions:

1. Since all the definitions depend on the assumptions that the involved agents are active, change of the agent status to inactive (e.g., agent failure) during system transitions would mean automatic update of *Attached, Responsible* and *Assigned* by removing all those records that refer to the failed agents;

2. In situations when the involved agents remain active during the system transitions, the above definitions do not forbid changing the actual relationships between the agents and the goals. In other words, the agents can be reattached, goal responsibility can be redistributed, and goals can be reassigned among the active agents.

Let us explicitly define multi-agent systems that support the dynamic reconfiguration described in the latter observation. Specifically, these are the systems that allow redistributing (unassigned) goals to different responsible agents or reattaching (unassigned) agents to different supervisor agents. The multi-robotic cleaning system that we have used as the running example is an instance of such systems.

Definition 17 Reconfigurable agent system *A MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$ *is reconfigurable if it is structured, open, and supports agent attachment, goal responsibility, and goal assignment. Moreover, the following properties hold*

(17.1)
$$\begin{aligned}\forall \sigma : \Sigma,\ g : \mathcal{G},\ : a_1, a_2 : \mathcal{A}.\ & (g \mapsto a_1) \in \mathit{Responsible}(\sigma)\ \wedge \\ & \mathit{gtype}(g) \in \mathit{AS\_goals}(\mathit{atype}(a\_2))\ \wedge \\ & \neg(\exists a_3 : \mathcal{A}.\ (g \mapsto a_1 \mapsto a_3)) \in \mathit{Assigned}(\sigma))\ \Rightarrow \\ & \quad \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (g \mapsto a_2) \in \mathit{Responsible}(\sigma')\end{aligned}$$

and

$$(17.2) \qquad \forall \sigma : \Sigma,\ a_1, a_2, a_3 : \mathcal{A}.\ (a_1 \mapsto a_2) \in \mathit{Attached}(\sigma)\ \wedge \\ (\mathit{atype}(a_3) \mapsto \mathit{atype}(a_2)) \in \mathit{A\_Sub}\ \wedge \\ \neg(\exists g : \mathcal{G}.\ (g \mapsto a_1 \mapsto a_2)) \in \mathit{Assigned}(\sigma))\ \Rightarrow \\ \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (a_3 \mapsto a_2) \in \mathit{Attached}(\sigma')$$

In our first definition of a goal-oriented multi-agent system, we required that any system goal is reachable from some initial system state. For a reconfigurable MAGSTS system, we can formulate and prove a stronger property: *"Any goal that is not yet reached at any (non-final) system state is reachable."*

Theorem 1 Goal reachability in a reconfigurable agent system. *For a reconfigurable MAGSTS system* $(\mathcal{G}, \Sigma, \mathit{Init}, \mathit{Trans}, \mathit{GMap}, \mathcal{A}, \mathit{Active})$*, the following property is true:*

$$\forall \sigma : \Sigma,\ g : \mathcal{G}.\ \sigma \in \mathbf{dom}(\mathit{Trans})\ \wedge\ \sigma \notin \mathit{GMap}(g)\ \Rightarrow \\ \exists \sigma' : \Sigma.\ (\sigma \mapsto \sigma') \in \mathit{Trans}^+\ \wedge \sigma' \in \mathit{GMap}(g).$$

Proof 1 *Let us consider an arbitrary state* $\sigma : \Sigma$*, such that* $\sigma \in \mathbf{dom}(\mathit{Trans})$*, and an arbitrary goal* $g : \mathcal{G}$*, such that* $\sigma \notin \mathit{GMap}(g)$*. Moreover, as the worst case scenario, let us assume that there is no single active agent able to supervise this goal nor single active agent able to accomplish it. Formally,*

$$\neg(\exists a : \mathcal{A}.\ a \in \mathit{Active}(\sigma)\ \wedge\ \mathit{gtype}(g) \in \mathit{AS\_goals}(\mathit{atype}(a)))$$

and

$$\neg(\exists a : \mathcal{A}.\ a \in \mathit{Active}(\sigma)\ \wedge\ \mathit{gtype}(g) \in \mathit{A\_goals}(\mathit{atype}(a))).$$

According to Definition 8 of a typed MAGSTS system, there should exists an agent able to supervise the goal g as well as a one able to accomplish it. Moreover, since our system is open, the second property of an open MAGSTS system (Definition 7) states a possibility to activate any agent at an arbitrary moment.

Let $a\_super : \mathcal{A}$*, such that*

$$a\_super \notin \mathit{Active}(\sigma) \qquad \textit{and} \qquad \mathit{gtype}(g) \in \mathit{AS\_goals}(\mathit{atype}(a\_super))$$

be an agent able to supervise the goal g. Moreover, let $\sigma_1 : \Sigma$*, such that* $(\sigma \mapsto \sigma_1) \in \mathit{Trans}$ *and* $\mathit{Active}(\sigma_1) = \mathit{Active}(\sigma) \cup \{a\_super\}$*, be a next state where*

this agent is activated. The existence of such state is required by Definition 7.

In a similar way, we activate an agent for accomplishing the goal g, a_worker, such that

$$a\_worker \notin Active(\sigma) \qquad and \qquad gtype(g) \in A\_goals(atype(a\_worker))$$

in a next state, σ_2, such that

$$(\sigma_1 \mapsto \sigma_2) \in Trans \qquad and \qquad Active(\sigma_2) = Active(\sigma_1) \cup \{a\_worker\}.$$

Relying on the definitions of agent attachment, goal responsibility and goal assignment (Definitions 14, 15 and 16), we can construct a chain of further states σ_3, σ_4, σ_5, where a_super and $a\_workers$ become attached, a_super takes responsibility for the goal g, and a_worker gets assigned the goal g respectively. There could be also as many as necessary intermediate state transitions where the statuses of a_super and a_worker are unaffected.

Finally, the definition of agent assignment (more specifically, the last consequent of property (16.2) of Definition 16) also connects this notion with goal reachability. Namely, for any state where a particular agent is assigned a specific goal, there exists a possible subsequent state where this goal is reached. Since the state σ_5 satisfies these criteria, we can claim that there exists a state, σ', such that

$$(\sigma_5 \mapsto \sigma') \in Trans \ \wedge \ \sigma' \in GMap(g).$$

By transitivity, we proved that

$$(\sigma \mapsto \sigma') \in Trans^{+} \ \wedge \sigma' \in GMap(g),$$

which is exactly what the theorem states.

In a similar manner, we can construct proofs for less adverse cases involving agent failures (i.e., becoming inactive), which in turn lead to specific agents becoming unattached, unassigned or specific goals losing the supervisors responsible for their completion.

To complete the proof, we have also consider the system states when the system has all the active agents needed to achieve a particular unreached goal, however the specific agent attachment and goal responsibility distribution has to be adjusted first. In other words, the system has to be reconfigured before proceeding.

Let us again consider an arbitrary state $\sigma : \Sigma$, such that $\sigma \in \mathbf{dom}(Trans)$, and an arbitrary goal $g : \mathcal{G}$, such that

$$\sigma \notin GMap(g) \qquad and \qquad \neg(\exists a_1, a_2 : \mathcal{A}.\ (g \mapsto a_1 \mapsto a_2) \in Assigned(\sigma)).$$

The completion of g in the state σ is responsibility of the agent a_super, i.e., $(g \mapsto a\_super) \in Responsible(\sigma)$.

Moreover, there exist the agents a_other and a_worker such that

$$(a\_super \mapsto a\_worker) \in Attached(\sigma) \ \ and \ \ gtype(g) \in A\_goals(atype(a\_worker)).$$

In other words, a_worker is attached to a_other and is able to accomplish the goal g. This also implies that a_other is able to supervise the goal g.

Relying on the definition of a reconfigurable multi-agent system (Definition 17), we can state that exists a next state σ'', where the agent a_worker is now attached to the new supervisor a_super, i.e.,

$$(g \mapsto a\_super) \in Responsible(\sigma'') \ \ and \, a\_super \mapsto a\_worker) \in Attached(\sigma'').$$

Alternatively, we can state that exists a next state σ'', where the responsibility of completing g is now moved to the new supervisor a_other, i.e.,

$$(g \mapsto a\_other) \in Responsible(\sigma'') \ \ and \ \ (a\_other \mapsto a\_worker) \in Attached(\sigma'').$$

In both cases, we can proceed by assigning the goal g to the agent a_worker and completing the goal as described above. In other words, we can show that there is the state $\sigma' : \Sigma$ such that

$$(\sigma'' \mapsto \sigma') \in Trans^+ \ \wedge \ \sigma' \in GMap(g).$$

By transitivity, we get that

$$(\sigma \mapsto \sigma') \in Trans^+ \ \wedge \sigma' \in GMap(g),$$

which is again exactly what we needed to prove.

□

The theorem requires for a multi-agent system to be open, which is not always the case in practice. In general, even without the openness assumption, we can still demonstrate goal reachability provided there always exists at least one agent which is able to accomplish this goal as well as one agent which is able to supervise it. The incorporated reconfigurability mechanisms will be then still sufficient to enable completion of the goal. Alternatively, we can quantitatively assess (based on the given agent failure and service rates) goal reachability using probabilistic model checking. In our previous work [32], we have employed such techniques for quantitative assessment of goal-oriented multi-agent systems using the PRISM probabilistic model checker. To enable probabilistic analysis of system models in PRISM, we have relied on our continuous-time probabilistic extension of the Event-B framework [33].

4 Formal Development of a Goal-Oriented MAS in Event-B

In the previous section we presented a general theory for reasoning about goal-oriented multi-agent state transition systems with incorporated reconfiguration mechanisms. There are many formalisms that support modelling and verification of state transition systems. In this section we will briefly overview one of them – Event-B – and present a number guidelines demonstrating how the notions defined above can be easily transferred and incorporated in Event-B. In a sense, by doing this we show a possible instantiation of our general theory in a concrete formalism. In turn, this should facilitate formal development of a goal-oriented MAS in Event-B.

4.1 Event-B: Background

Event-B is a state-based framework that promotes the correct-by-construction approach to system development and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [1]. An Abstract State Machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The variables are strongly typed by the constraining predicates that together with other important properties of the systems are defined in the model *invariants*. Usually, a machine has an accompanying component, called *context*, which includes user-defined sets, constants and their properties given as a list of model axioms.

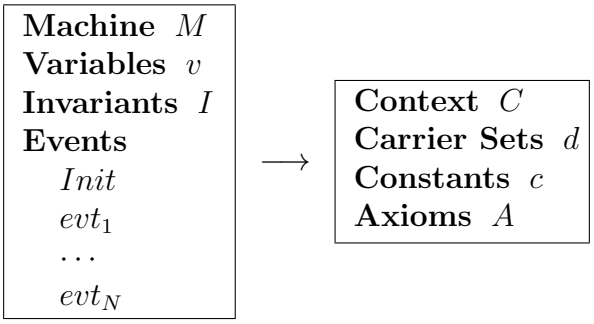

Figure 2: Event-B machine and context

A general form for Event-B models is given in Fig. 2. The machine is uniquely identified by its name M. The state variables, v, are declared in the **Variables** clause and initialised in the $Init$ event. The variables are strongly typed by the constraining predicates I given in the **Invariants** clause. The invariant clause might also contain other predicates defining properties (e.g., safety invariants) that should be preserved during system execution.

The dynamic behaviour of the system is defined by a set of atomic *events*. Generally, an event has the following form:

$$e \mathrel{\widehat{=}} \textbf{any } a \textbf{ where } G_e \textbf{ then } R_e \textbf{ end},$$

where e is the event's name, a is the list of local variables, the *guard* G_e is a predicate over the local variables of the event and the state variables of the system. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment over the system variables. In Event-B, an assignment represents a corresponding next-state relation R_e. The guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

If an event does not have local variables, it can be described simply as:

$$e \mathrel{\widehat{=}} \textbf{when } G_e \textbf{ then } R_e \textbf{ end}.$$

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps, we gradually reduce nondeterminism and introduce detailed design decisions. In particular, we can add new events, split events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. When data refinement is performed, we define *gluing invariants* as a part of the invariants of the refined machine. They define the relationship between the abstract and concrete variables.

Often a refinement step introduces new events and variables into the abstract specification. The new events correspond to the stuttering steps that are not visible at the abstract level, i.e., they refine implicit skip. To guarantee that the refined specification preserves the global behaviour of the abstract machine, we should demonstrate that the newly introduced events converge. To prove it, we need to define a variant an expression over a finite subset of natural numbers and show that the execution of new events decreases it. Sometimes, convergence of an event cannot be proved due to a high level of non-determinism. Then the event obtains the status *anticipated*. This obliges the designer to prove at some later refinement step, that the event indeed converges.

The consistency of Event-B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps, is demonstrated by discharging a number of verification conditions – proof obligations. The Rodin platform [27] provides an automated support for formal modelling and verification in Event-B. In particular, it automatically generates the required proof obligations and attempts to discharge them. The remaining unproven conditions can be dealt with by using the provided interactive provers.

Context C0
Sets $Goals$, $Agents$
Constants G_graph, $GraphClosure$, $Subgoals$
Axioms
axm1: $Goals \neq \varnothing$
axm2: $Agents \neq \varnothing$
axm3: $G\_graph \in Goals \leftrightarrow Goals$
axm4: $GraphClosure \in Goals \leftrightarrow Goals$
axm5: $G\_graph \subseteq GraphClosure$
axm6: $\forall g1, g2 \cdot (g1 \mapsto g2) \in G\_graph \Leftrightarrow$
$(g1 \mapsto g2) \in G\_graph \vee (\exists g3 \cdot (g1 \mapsto g3) \in G\_graph \wedge (g3 \mapsto g2) \in GraphClosure)$
axm7: $\forall g \cdot g \in Goals \Rightarrow (g \mapsto g) \notin GraphClosure$
axm8: $\forall g \cdot g \in Goals \Rightarrow Subgoals(g) = \{g1 \mid (g \mapsto g1) \in G\_graph\}$
...

Figure 3: Context C0

4.2 A Goal-Oriented MAS in Event-B

To formally develop a multi-agent system based on the theory presented in the previous section, we have to translate or represent the introduced notions and definitions in terms of the corresponding Event-B elements. Below we present our guidelines for such a translation.

Event-B separates the static and dynamic parts of a model, putting them into distinct yet dependent components called a context and a machine. Similarly, for our theory, we must first distinguish static and dynamic concepts and then do, if necessary, a further classification of them to translate the resulting cases into specific Event-B elements. The obvious criterion for such a separation is the direct dependence of the concept in question on the type Σ denoting the system state space. To be precise, if its type depends on Σ (see Table 1 for particular cases of such dependence), we consider a concept is a dynamic one and it must be represented as one of the elements (e.g., state variables or events) in the model machine component(s). Otherwise, it is considered static and becomes one of the elements of a model context.

Representation of static concepts of goal-oriented MAS in Event-B. Using the above criterion, the static notions of our theory include the types for all possible goals and agents ($\mathcal{G}$ and $\mathcal{A}$) and their types ($GType$ and $AType$) as well as different structures defining various classifications and interdependencies between elements of these types. The latter include G_graph, GT_graph, $atype$, $gtype$, A_goals, G_agents, AG_Rel, A_Sub, AS_goals, GS_agents, and so on.

We introduce static notions as sets and constants of a model context and define their properties as a number of context axioms. For instance, the following excerpt (Fig.3) defines the sets $Goals$ and $Agents$ (i.e., $\mathcal{G}$ and $\mathcal{A}$) as well as the constants G_graph, $GraphClosure$ (i.e., G_graph^+), and $Subgoals$.

Note that, since both our theory and Event-B are based set theory and

Context C1 **extends** C0
Sets $GType$, $AType$
Constants $atype$, $gtype$, GT_graph, $ROBOT$, $B\_STATION$, $ZONE\_CLEANING$, AG_Rel
$SECTOR\_CLEANING$, $Robots$, $BStations$, $ZCleaning$, $SCleaning$
Axioms
axm1: $ZCleaning \subseteq Goals \ \wedge SCleaning \subseteq Goals$
axm2: $Robots \subseteq Goals \ \wedge BStations \subseteq Goals$
axm3: $(Goals = ZCleaning \cup SCleaning) \ \wedge \ (ZCleaning \cap SCleaning = \varnothing)$
axm4: $(Agents = Robots \cup BStations) \ \wedge \ (Robots \cap BStations = \varnothing)$
axm5: $GType = \{ZONE\_CLEANING, SECTOR\_CLEANING\}$
axm6: $AType = \{ROBOT, B\_STATION\}$
axm7: $gtype \in Goals \rightarrow GType$
axm8: $atype \in Agents \rightarrow AType$
axm9: $gtype[Robots] = \{ROBOT\}$
axm10: $gtype[BStations] = \{B\_STATION\}$
axm11: $atype[ZCleaning] = \{ZONE\_CLEANING\}$
axm12: $atype[SCleaning] = \{SECTOR\_CLEANING\}$
axm13: $GT\_graph \in GType \leftrightarrow GType$
axm14: $GT\_graph = \{ZONE\_CLEANING \mapsto SECTOR\_CLEANING\}$
axm15: $AG\_Rel \in AType \leftrightarrow GType$
axm16: $AG\_Rel = \{B\_STATION \mapsto ZONE\_CLEANING, ROBOT \mapsto SECTOR\_CLEANING\}$
...

Figure 4: Context C1

predicate calculus, the considered definitions are translated in a rather straightforward way. Such a translation gives a generic context that may be used for modelling of a class of suitable systems or, alternatively, used in very abstract models which are later refined by constraining (instantiating) the defined structures for concrete cases.

In the following excerpt of a refined context (Fig.4), we constrain the abstract definitions of *Goals* and *Agents* to those of our running example (the multi-robotic cleaning system described in Section 2), as defined in the axioms 1–4. We also give concrete definitions for the introduced types *GType* and *AType* (axioms 5–6), the functions *gtype* and *atype* (axioms 7–12), and the relation structures *GT_graph* and *AG_Rel* (axioms 13–16). It can be easily demonstrated that these axioms are proper instantiations of their general definitions given in Definition 8 (*atype, gtype*), Definition 9 (*AG_Rel*) and Definition 11 (*GT_graph*).

Representation of dynamic concepts of goal-oriented MAS in Event-B. In general, the system dynamics (formalised as state transitions on state space Σ constrained by the relation $Trans$) is represented as machine events in Event-B. However, various introduced concepts that affect this dynamics (e.g., connecting particular state transitions with goal and agent structures, supervision and reconfiguration mechanisms, the properties to be preserved, etc.) can be represented as different elements of an Event-B machine, such as model variables, invariants, predicate expressions, or specific events. Table 1 gives a summary of such possible representations.

For instance, a number of dynamic system attributes (such as *Active*,

Table 1: Translation guidelines

Theory definition	Event-B counterpart
$Trans : \Sigma \leftrightarrow \Sigma$	initialisation and events of a machine
functions of the form $\Sigma \to T$	machine variables of the type T
$GMap(g)$, $SGMap(g) : \mathcal{G} \to \mathcal{P}(\Sigma)$	a predicate over machine variables
a property of the form $\forall \sigma : \Sigma.\ P(\sigma)$	a machine invariant
a property of the form $\forall \sigma : \Sigma.\ P(\sigma) \Rightarrow \exists \sigma'.\ \sigma \mapsto \sigma' \in Trans \wedge R(\sigma, \sigma')$	a specific machine event

Machine M1
Sees C1
Variables $Active, Attached, ...$
Invariants
`inv1:` $Active \in \mathbb{P}(Agents)$
`inv2:` $Attached \in \mathbb{P}(Agents \times Agents)$
`inv3:` $\forall a1, a2 \cdot\ (a1 \mapsto a2) \in Attached \ \Rightarrow\ a1 \in Active \wedge a2 \in Active$
`inv4:` $\forall a1, a2 \cdot\ (a1 \mapsto a2) \in Attached \ \Rightarrow\ atype(a1) \mapsto atype(a2) \in A\_Sub$
`inv5:` $\forall a1, a2 \cdot\ (a1 \mapsto a2) \in Attached \Rightarrow \neg(\exists a3.\ a3 \neq a1 \wedge (a3 \mapsto a2) \in Attached)$
...
Events
...
end

Figure 5: Model variables and invariants

Attached, *Responsible*, *Assigned*, etc.) are formalised as functions of the form $\Sigma \to T$. They can be naturally represented as model variables of the type T. In their definitions, these attributes are usually associated with some defining properties that are supposed to be preserved in each reachable state. These properties then become invariants of the resulting Event-B model.

As an example, let us consider the definition of agent attachment (Definition 14). It introduces a dynamic attribute (function) $Attached : \Sigma \to \mathcal{P}(\mathcal{A} \times \mathcal{A})$ with the following property (14.2)

$$\begin{array}{l} \forall \sigma : \Sigma,\ a_1, a_2 : \mathcal{A}.\ (a_1 \mapsto a_2) \in Attached(\sigma)\ \Rightarrow \\ \quad a_1 \in Active(\sigma)\ \wedge\ a_2 \in Active(\sigma)\ \wedge\ atype(a_1) \mapsto atype(a_1) \in A\_Sub\ \wedge \\ \quad \neg(\exists a_3 : \mathcal{A}.\ a_3 \neq a_1\ \wedge\ (a_3 \mapsto a_2)\ \in\ Attached(\sigma)). \end{array}$$

In its turn, *Attached* depends on another dynamic attribute (state variable) *Active* defined as $Active : \Sigma \to \mathcal{P}(\mathcal{A})$. The following excerpt from an Event-B machine (Fig.5) demonstrates how both *Active* and *Attached* can be represented.

Another kind of dynamic properties is often expressed in the form

$$\forall \sigma : \Sigma.\ P(\sigma) \Rightarrow \exists \sigma'.\ \sigma \mapsto \sigma' \in Trans \wedge R(\sigma, \sigma').$$

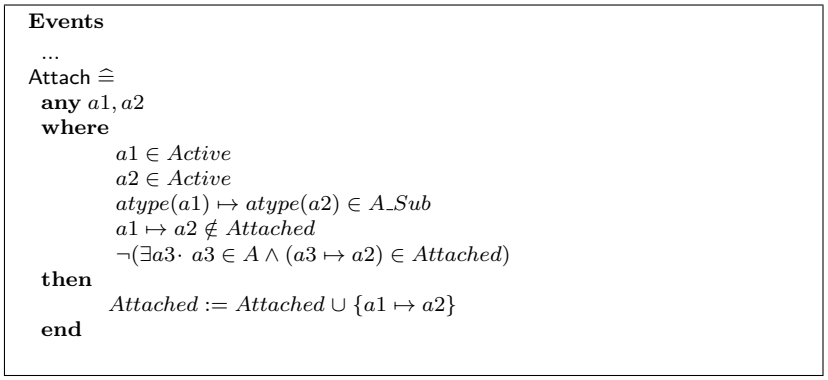

Figure 6: Model events

Essentially, such properties require existence a particular kind of state transitions in the system. Since state transitions are represented as model events in Event-B, this is an indication that a specific model event should be constructed, thus implementing the given property.

Let us go back to the definition of agent attachment (Definition 14). The last definition property (14.3) requires that

$$\begin{array}{l} \forall \sigma : \Sigma,\ a_1, a_2 : \mathcal{A}.\ a_1 \in \mathit{Active}(\sigma)\ \wedge\ a_2 \in \mathit{Active}(\sigma)\ \wedge \\ \quad \mathit{atype}(a_1) \mapsto \mathit{atype}(a_2) \in \mathit{A\_Sub}\ \wedge\ \neg(\exists a_1' : \mathcal{A}.\ (a_1' \mapsto a_2)\ \in\ \mathit{Attached}(\sigma)) \\ \Rightarrow \\ \quad \exists(\sigma' : \Sigma).\ (\sigma \mapsto \sigma') \in \mathit{Trans}\ \wedge\ (a_1 \mapsto a_2) \in \mathit{Attached}(\sigma'). \end{array}$$

As a result of event construction, the left hand side of implication then becomes the event guard, while the right hand side defines the required action of the event (see Fig.6).

Finally, in our formalisation the functions $GMap : \mathcal{G} \to \mathcal{P}(\Sigma)$ and $SGMap : \mathcal{G} \to \mathcal{P}(\Sigma)$ relate goals with specific states where goals (or some expressions on their subgoals) are considered as reached. In Event-B, we can model these functions as particular predicates on state variables storing such information about reached goals. Often, such information is partitioned (stored on distinct variables) according to the involved goal type.

Let us consider again our example of the system with cleaning robots and coordinating base stations. We have two types of goals – zone cleaning and sector cleaning – which are responsibilities of base stations and robots respectively. The information about reached goals can be stored in two distinct boolean array variables: *zone_is_cleaned* for the goals in $CleaningZones$, and *sector_is_cleaned* for the goals in $CleaningSectors$. Then $GMap(g)$ can be represented as the predicate $\mathit{zone\_is\_cleaned}(g) = TRUE$, if $g \in CleaningZones$, and $\mathit{sector\_is\_cleaned}(g) = TRUE$ otherwise.

Recall that $SGMap(g)$ serves as the precondition for reaching the goal g, while the goal completion of g may not be officially recorded yet. In our

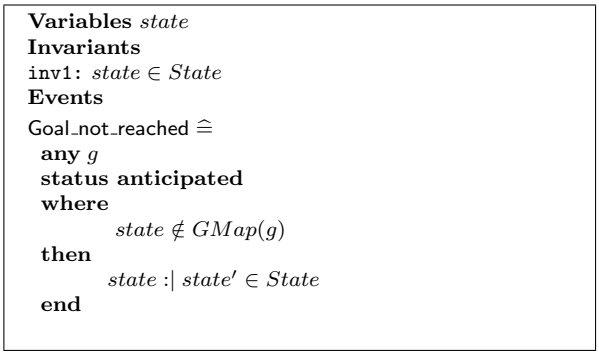

Figure 7: Anticipated goal reachability

example system, $SGMap(g)$ for $g \in CleaningSectors$ may be represented, e.g., as

$$CoverageSensor(r \mapsto s) = TRUE,$$

for some robot r and sector s, indicating that the whole sector area has been covered by the cleaning robot r, which may not yet reacted on that.

Goal reachability . In Definition 1, we define a goal-oriented multi-agent system as such that has ability to reach any of its goal from its initial states:

$$\forall g : \mathcal{G}.\ \exists \sigma, \sigma' : \Sigma.\ \sigma \in Init \ \wedge\ (\sigma \mapsto \sigma') \in Trans^* \ \wedge\ \sigma' \in GMap(g).$$

How can we enforce this property in Event-B? One possibility is to start with a very abstract system with a single event (see Fig.7).

Here the single state variable *state* is completely non-deterministically updated in the event *Goal_not_reached*. The anticipated status of the event indicates that we promise to prove convergence of this event, thus showing reachability of any system goal. The actual proof of such convergence is postponed until some later refined model, which has enough implementation details prove overall convergence based on a formulated variant expression.

Alternatively, we can rely on ProB, a model checker for Event-B, and verify goal reachability by formulating and checking the corresponding temporal logic property for the considered system model.

5 Related Work and Conclusions

Related Work. The field of design of multi-agent systems has considerable evolved over the last decade. Surveying the literature on MAS reveals a significant amount of research devoted to different agent organisation concepts, agent specification languages and platforms, modelling and verification agent behaviour, etc. The resulting approaches vary significantly in terms

of the covered topics, such as agent interoperability, communication, roles, goals and beliefs. Below we outline only a few works most relevant to our research.

The Tropos methodology [6] supports analysis and design in the development of agent-based software systems. UML diagrams are used to represent the system goals, agents, their capabilities and interdependencies, as well as system properties and agent interactions. An extension of this work [21] also supports modelling of agent errors and recovery activities.

Another proposed methodology - Multi-Agent System Engineering (MaSe) [8] – guides the designer through the software lifecycle of a multi-agent system. It allows graphically represent the system goals, the associated use cases and agent roles. Finite state automata are used to express communications between agent classes. The accompanied tool, the Agent Tool, supports the agent system development following the MaSe methodology. An extension of this work, Organization-based MaSE (O-MaSe) [9], provides a mechanism for defining agent interactions with the environment via external actors as well as defining the interaction protocols between the system and the actors. O-MaSE makes use of UML class diagrams and does not support formal notation.

Formal modelling of agent systems has been undertaken by [31, 30, 28, 29]. The authors have proposed an extension of the UNITY framework to explicitly define such concepts as mobility and context-awareness. The mobile UNITY [31] extension proposes the notation to express mobile computations and a logic for reasoning about components temporal properties. It also supports formal reasoning about mobile components and their behaviour. On the other hand, the Context UNITY extension [29] formalises context-aware computing, with the proposed notation to represent the system context. The sensed aspects of the environment are used by the system to adjust its behaviour. In our formalisation we have pursued a different goal – we aimed at formally guaranteeing that the specified agent behaviour with the incorporated reconfiguration mechanisms facilitates achieving the defined system goals.

Formal modelling of fault tolerant MAS in Event-B has been also undertaken by Ball and Butler [3]. They have proposed a number of informally described patterns that allow the designers to incorporate well-known (static) fault tolerance mechanisms into formal models. In our approach we consider fault tolerance as a part of ensuring resilience of MAS. Moreover, we have formalised a more advanced fault tolerance scheme that relies on goal reallocation and dynamic reconfiguration to guarantee goal reachability.

The use of model checking techniques for reasoning about MAS properties has been actively researched as well (see, e.g., [4, 5, 17, 11, 18]). In particular, [5] presents a framework for verification of agent programs against BDI (belief-desire-intention) agent specifications. In the proposed approach,

an agent system is first programmed using the logic-based agent oriented programming language AgentSpeak(F). Then the AgentSpeak(F) programs are translated into Promela – the specification language of the SPIN model checker – to verify the resulting system. Ferrari et al. [10] describe a verification of π-calculus based process algebra for mobile agents, while [18] presents modelling of fault-tolerant agents by stochastic Petri nets. The paper [17] describes the symbolic model checker MCMAS, specifically tailored for verification of MAS. The MCMAS tool takes as inputs models describing both agents and working environment of a multi-agent system and applies the epistemic logic to analyse it. However, model checking approaches typically suffer from the state space explosion problem, which is especially acute for large systems. As demonstrated by the proposed guidelines, our formalisation can be easily represented in the Event-B formalism. Since Event-B is based on theorem proving, this would help to avoid the mentioned problem.

The foundational work on *goal-oriented development* has been done by van Lamsweerde [7, 34, 36]. The proposed KAOS framework [7] provides a goal-oriented approach for requirements modelling, specification, and analysis, to address both functional and non-functional system requirements. Based on the KAOS framework, Lamsweerde [35] has proposed a method for deriving the software architecture from its requirements. Specifically, according to the method, the software specification is developed from the requirements which is then used to build the architectural design. The design is based with consecutive refinements, which take into account constraints and non-functional goals. The KAOS approach is supported by the GRAIL tool [7].

Over the last decade the goal-oriented approach has also received several extensions that allow the designers to link it with formal modelling [14, 25]. In particular, the work [14] presents the technique of translating KAOS operational models into event-based tabular specifications that can be then analysed by SCR* toolset [12]. The technique consists of a number of transformation steps each of which solves semantic, structural or syntactic dereferences between the KAOS and SCR (Software Cost Reduction) languages.

Significant amount of research has been devoted for translating formal specifications of software operations built according to the KAOS goal-oriented method into event-based transition systems. For example, the work [16] presents an approach to use the formal analysis capabilities of LTSA (Labelled Transition System Analyser) to analyse and animate KAOS operational models. The mapping allows designers to translate goal-oriented operational requirements into a black-box event-based model of the software behaviour expressed in a formalism appropriate to reason about behaviours at the architectural level.

One of the first attempt to bridge KAOS operations with B specifica-

tion was presented in [26]. More recently, the study to formalise KAOS in Event-B was attempted in [2]. The paper proposes a constructive approach that allows to link high-level system requirements expressed as linear temporal logic formulae to the corresponding Event-B elements. The notion of a triggered event is used to translate time operators that are used in KAOS models. Similar, Matoussi et al. [19, 20] describe works on coupling requirements engineering methods with formal methods. In contrast, in our work we have relied on goals to facilitate structuring of the system behaviour, while connecting them with agent collaboration and system reconfiguration mechanisms.

In our previous work on goal reachability and agent collaboration, we have investigated a colony of ants [13]. We have formalised the behaviour of cooperative ants in Event-B and verified by proofs that the desired system-level properties become achievable via agent collaboration. The proposed approach allows the designers to rigorously define constraints on the environment and the ant behaviour at different abstraction levels and systematically explore the relationships between system-level goals, environment and autonomous ants.

Conclusions. The main paper contribution is the proposed theoretical study of resilient goal-oriented multi-agent systems. The formalisation gradually defines the main notions of such systems, together with their intricate relationships between different agent and goal structures as well as the incorporated dynamic reconfiguration mechanisms. The latter allow the system to become more resilient with respect to the system goals and also more collaborative with respect to the involved system agents. The final theorem is proved to formally demonstrate that all the introduced notions and mechanisms are sufficient to ensure goal reachability in such a system. The presented work is based on our experience in formal modelling and verification of resilient goal-oriented multi-agent systems, see, e.g., [22, 23, 32, 13].

There is a number of features and properties of such systems that were left out from this formalisation. For instance, it would be interesting to more deeply investigate how the information about goal reachability is stored (distinguishing the local and global knowledge) and later propagated from agents to their supervisors and beyond. This issue is also directly related with the representation of different levels of perception that some goal is now completed, how this perception is propagated through the agent and goal hierarchies, the order of this propagation and delays related with it. In turn, such knowledge can be used to make the system reconfiguration mechanisms more efficient by, e.g, avoiding in some cases to redo an already accomplished goal after the supervisor agent responsible for this goal has failed. The listed topics constitute the basis for our future work in this research area.

References

[1] J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[2] Benjamin Aziz, Alvaro Arenas, Juan Bicarregui, Christophe Ponsard, and Philippe Massonet. From goal-oriented requirements to event-b specifications. In *First NASA Formal Methods Symposium - NFM 2009*, pages 96–105, 2009.

[3] Elisabeth Ball and Michael Butler. Event-B Patterns for Specifying Fault-Tolerance in Multi-agent Interaction. In *Methods, Models and Tools for Fault Tolerance*, pages 104–129. Springer, 2009.

[4] Rafael Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model Checking AgentSpeak. In *AAMAS 2003*, pages 409–416. ACM Press, 2003.

[5] Rafael H. Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying Multi-agent Programs by Model Checking. *Autonomous Agents and Multi-Agent Systems*, 12(2):239–256, 2006.

[6] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[7] Robert Darimont, Emmanuelle Delor, Philippe Massonet, and Axel van Lamsweerde. GRAIL/KAOS: an environment for goal-driven requirements engineering. In *Proceedings of the 19th International Conference on Software Engineering*, pages 612–613. ACM, 1997.

[8] Scott A. DeLoach. The mase methodology. 11:107–125, 2004.

[9] Scott A. DeLoach and Juan C. García-Ojeda. O-mase: a customisable approach to designing and building complex, adaptive multi-agent systems. *IJAOSE*, 4(3):244–280, 2010.

[10] Gian-Luigi Ferrari, Stefania Gnesi, Ugo Montanari, and Marco Pistore. A model-checking verification environment for mobile processes. *ACM Trans. Softw. Eng. Methodol.*, 12(4):440–473, October 2003.

[11] Jianye Hao, Songzheng Song, Yang Liu, Jun Sun, Lin Gui, Jin Song Dong, and Ho-fung Leung. Probabilistic Model Checking Multi-agent Behaviors in Dispersion Games Using Counter Abstraction. In *PRIMA 2012*, volume 7455 of *LNCS*, pages 16–30. Springer, 2012.

[12] Constance L. Heitmeyer, James Kirby, Bruce G. Labaw, and Ramesh Bharadwaj. Scr*: A toolset for specifying and analyzing software requirements. In *Computer Aided Verification, 10th International Conference, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 526–531. Springer, 1998.

[13] Linas Laibinis, Elena Troubitsyna, Zeineb Graja, Frédéric Migeon, and Ahmed Hadj Kacem. Formal Modelling and Verification of Cooperative Ant Behaviour in Event-B. In *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014*, volume 8702 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2014.

[14] Renaud De Landtsheer, Emmanuel Letier, and Axel van Lamsweerde. Deriving tabular event-based specifications from goal-oriented requirements models. *Requir. Eng.*, 9(2):104–120, 2004.

[15] J.-C. Laprie. From Dependability to Resilience. In *DSN 2008, Dependable systems and Networks*. IEEE Computer Society, 2008.

[16] Emmanuel Letier, Jeff Kramer, Jeff Magee, and Sebastián Uchitel. Deriving event-based transition systems from goal-oriented requirements models. *Autom. Softw. Eng.*, 15(2):175–206, 2008.

[17] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. MCMAS: A Model Checker for the Verification of Multi-Agent Systems. In *CAV 2009*, volume 5643 of *LNCS*, pages 682–688. Springer, 2009.

[18] Michael R. Lyu, Xinyu Chen, and Tsz Yeung Wong. Design and evaluation of a fault-tolerant mobile-agent system. *IEEE Intelligent Systems*, 19(5):32–38, 2004.

[19] Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau. A first attempt to express KAOS refinement patterns with event B. In *Abstract State Machines, B and Z, First International Conference, ABZ 2008*, page 338, 2008.

[20] Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau. A goal-based approach to guide the design of an abstract event-b specification. In *16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011*, pages 139–148, 2011.

[21] Mirko Morandini, Loris Penserini, and Anna Perini. Towards goal-oriented development of self-adaptive systems. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 9–16. ACM, 2008.

[22] Inna Pereverzeva, Elena Troubitsyna, and Linas Laibinis. A Case Study in Formal Development of a Fault Tolerant Multi-robotic System. In *Software Engineering for Resilient Systems - 4th International Workshop, SERENE 2012, Pisa, Italy, September 27-28, 2012. Proceedings*, volume 7527 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2012.

[23] Inna Pereverzeva, Elena Troubitsyna, and Linas Laibinis. Formal Development of Critical Multi-agent Systems: A Refinement Approach. In *2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012*, pages 156–161. IEEE, 2012.

[24] Inna Pereverzeva, Elena Troubitsyna, and Linas Laibinis. Formal Goal-Oriented Development of Resilient MAS in Event-B. In *Reliable Software Technologies - Ada-Europe 2012 - 17th Ada-Europe International Conference on Reliable Software Technologies, Stockholm, Sweden, June 11-15, 2012. Proceedings*, volume 7308 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2012.

[25] Christophe Ponsard, Gautier Dallons, and Massone Philippe. From Rigorous Requirements Engineering to Formal System Design of Safety-Critical Systems. In *ERCIM News (75)*, pages 22–23, 2008.

[26] Christophe Ponsard and Emmanuel Dieul. From requirements models to formal specifications in B. In *Proceedings of the CAISE*06 Workshop on Regulations Modelling and their Validation and Verification ReMo2V '06*, 2006.

[27] Rodin. Event-B Platform. online at http://www.event-b.org/.

[28] Gruia-Catalin Roman, Christine Julien, and Jamie Payton. A formal treatment of context-awareness. In *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *Lecture Notes in Computer Science*, pages 12–36. Springer, 2004.

[29] Gruia-Catalin Roman, Christine Julien, and Jamie Payton. Modeling adaptive behaviors in context UNITY. *Theor. Comput. Sci.*, 376(3):185–204, 2007.

[30] Gruia-Catalin Roman and Peter J. McCann. A notation and logic for mobile computing. *Formal Methods in System Design*, 20(1):47–68, 2002.

[31] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: reasoning and specification in mobile computing. *ACM Trans. Softw. Eng. Methodol.*, 6(3):250–282, 1997.

[32] Anton Tarasyuk, Inna Pereverzeva, Elena Troubitsyna, and Linas Laibinis. Formal development and quantitative assessment of a resilient multi-robotic system. In *Software Engineering for Resilient Systems, 5th International Workshop, SERENE 2013, Kiev, Ukraine, October 3-4, 2013. Proceedings*, volume 8166 of *Lecture Notes in Computer Science*, pages 109–124. Springer, 2013.

[33] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Integrating stochastic reasoning into Event-B development. *Formal Aspects of Computing*, 27(1):53–77, 2015.

[34] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *Requirements Engineering*, pages 249–263, 2001.

[35] Axel van Lamsweerde. From system goals to software architecture. In *Formal Methods for Software Architectures*, volume 2804 of *Lecture Notes in Computer Science*, pages 25–43. Springer, 2003.

[36] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-3207-7
ISSN 1239-1891