UiO : **Department of Informatics**
University of Oslo

The Cooperative Cleaners Case Study: Modelling and Analysis in Real-Time ABS

Silvia Lizeth Tapia Tarifa

Master's Thesis Autumn 2013

The Cooperative Cleaners Case Study: Modelling and Analysis in Real-Time ABS

Silvia Lizeth Tapia Tarifa

Autumn 2013

Abstract

Swarm robotics has been proposed as a way to organise decentralised systems in which a large number of similar *robots*, that are autonomous and relatively simple in their behaviour, are coordinated in a way that leads to an emergent useful behaviour. *The Cooperative Cleaners case study* is an example of swarm robotics. In this case study, a group of robots cooperate in order to reach the common goal of cleaning a dirty floor. A cleaning robot has a bounded amount of memory; it can only observe the state of the floor in its immediate surroundings and decide on its movement based on these observations. The cleaning robots use indirect communication based on sensing and signalling, and the desired goal of cleaning the whole floor is thus an emergent behaviour of the swarm. In this thesis we show how *the Cooperative Cleaners case study* can be modelled and analysed using the Real-Time ABS language. Real-Time ABS is a timed, abstract and behavioural specification language with a formal semantics and a Java-like syntax that targets concurrent, distributed and object-oriented systems. We use the features of Real-Time ABS in the development of our model of *the Cooperative Cleaners problem* in order to evaluate how well Real-Time ABS can be used to model and analyse the behaviour of systems from the swarm robotics domain. Our work shows that Real-Time ABS can facilitate the modelling of these kinds of systems, but due to the challenges introduced when formally modelling systems in this domain, it requires fairly advanced modelling skills to obtain an adequate model. However the analysis of these kinds of systems is currently restricted to simulations applied to small scenarios.

Preface

This master's thesis describes a work done at the Precise Modelling and Analysis (PMA) group within the Department of Informatics at the University of Oslo.

My task for this thesis was to model a case study from the swarm robotics domain using the ABS language. The ABS language was developed by the European Project HATS (Highly Adaptable and Trustworthy Software using Formal Models). The PMA group was one of the partners in this project.

I would like to thank my main supervisor and friend Einar Broch Johnsen for his priceless time, accurate advice and constant support in my long road towards a master's degree.

My thanks to everyone in the PMA group including the professors, researchers and PhD students. In particular I am grateful to Olaf Owe for his understanding and support.

Finally I would like to thank my parents Patricia and Walter, and my brothers Fabricio and Alex for their company despite the distance and for their solid faith and trust in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A multi-robot system is a distributed system in which robots communicate and coordinate their actions in order to achieve a common goal. Two significant characteristics of multi-robot system are concurrency and fault tolerance. *Swarm robotics* is one approach that typically emphasises the decentralised coordination of multi-robot systems which consist of a large numbers of mostly simple physical robots. This approach is inspired by the field of artificial swarm intelligence, as well as biological studies of insects, ants and other examples from nature, where swarm behaviour occurs. Unlike distributed robotic systems in general, swarm robotics emphasises a large number of robots, and promotes scalability, for example by using only local communication. Moreover, in swarm robotics, the desired collective behaviour should emerge from the interactions between the robots and the interactions of the robots with the environment.

The Cooperative Cleaners problem is a problem that requires several simple robots to clean a dirty connected region in $\mathbb{Z}^2$. These simple robots move on the dirty region. In this problem each robot has the ability to clean its location and their common goal is to collectively clean the given dirty region.

In this thesis we will model a case study of *the Cooperative Cleaners problem* using the modelling language ABS.

ABS (for abstract behavioural specification) [20] is a language for modelling object-oriented systems at an abstract, yet precise level. In [19] Hähnle emphasises that:

- ABS has a clear and simple concurrency model that permits synchronous as well as actor-style [1] asynchronous communication [21];
- ABS abstracts from specific implementation of datatype, but is a fully executable language and has code generators for other languages including Java and Maude [15];
- ABS targets software systems that are concurrent, distributed and object-oriented;
- ABS has a formal semantics; and
- ABS offers a wide variety of modelling options in one framework such us algebraic datatypes, functions, classes, concurrency, distribution, etc. This allows to select an appropriate modelling style for each modelled entity.

Real-Time ABS [9] extends the syntax and semantics of ABS in order to allow models with time dependent behaviour.

1.1 Problem Statement

The goal of this thesis is to develop a case study from the swarm robotics domain that exploits features of ABS such as concurrency, distribution and object-orientation in order to evaluate how well ABS can be used to model and analyse the behaviour of systems that are autonomous, decentralised and self-organised. *The Cooperative Cleaners case study* is an interesting example of swarm behaviour where a group of simple robots cooperate to clean a dirty region.

This thesis aims to answer the following questions.

1. How can Real-Time ABS be used to naturally model autonomous, decentralised and self-organised systems such as swarm robotics?
2. To what extent can the simulation tool of Real-Time ABS help to analyse the collective behaviour of such systems?

We answer these questions by developing *the Cooperative Cleaners case study* using Real-Time ABS and later analyse the behaviour of the resulting model by means of simulations.

We will come back to these questions in the conclusion of this thesis (Section 6.2).

1.2 Related Work

There is relatively little work on formally modelling and analysing multi-robot systems, in particular swarm robotics.

In [32] Winfield *et al.* present results of applying a proposed probabilistic finite state machine (PFSM) in the description of wireless networks connectivity and overall macroscopic behaviour of swarms. In [25] Kiriakidis and Gordon-Spears present a practical method for modelling and controlling reconfigurable teams of robots using discrete event systems to capture the collective variable behaviour of the team and later analyse it using model checking. In this work the authors consider robots that communicate with a coordinator and where the whole system behaves as a robotic group. In [7] Alur *et al.* develop a case study for formally modelling multi-robot coordination systems using hybrid automata and the tool HYTECH for the symbolic analysis of the model. In this paper the authors focus on the design of communication and control strategies for a team of autonomous, mobile robots.

The works presented above are more interested in the representation of the swarm in a macroscopic view where models reproduce the swarm system as a single representation, abstracting from the behaviour of the individual robots. In contrast, this thesis has a microscopic view, because we are interested in capturing the behaviour of each individual robot in an abstract model and analyse the collective behaviour of the resulting swarm.

The Cooperative Cleaners problem has been introduced and studied in a series of papers by Bruckstein *et al.* [4–6, 31]. In [31] the authors describe a static version of the problem. In the static version of the problem, the shape of the dirty area of the floor does not change on its own and its area only decreases when one of the robots clean a portion of it. In this paper the authors propose CLEAN, a cleaning protocol to ensure that the cleaning robots stop only upon completing their common goal of cleaning a dirty floor. The authors give an informal description of the distributed algorithm and an argument for the correctness of the whole system; additionally they provide experimental results obtained by means of computer simulations. In [5,6], the authors describe a dynamic version of the problem which involves a deterministic evolution of the floor, simulating a spreading contamination, in this case the shape of the dirty area deterministically increases from time to time. As in the static version, the authors provide a modification of the cleaning protocol for this dynamic environment together with its hand written analysis and experimental results by means of computer simulations. Vain *et al.* have developed a formal model of the cooperative cleaners in timed automata and use Uppaal [26] to show correctness properties for the dynamic cooperative cleaners using a setup with 16 tiles and two robots [30]. For simplicity in this thesis we focus on the analysis of an executable model for the static version of the problem as defined in [31].

1.3 Structure of this Thesis

The rest of this thesis is organised as follows:

- Chapter 2 introduces the Cooperative Cleaners problem.
- Chapter 3 provides a tutorial-like introduction to ABS and its extension with real-time.
- Chapter 4 explains in detail how the Cooperative Cleaners can be modelled in Real-Time ABS.
- Chapter 5 discusses the insights and results we obtain by analysing our model.
- Chapter 6 summarises the results in the context of the problem statement of Section 1.1 and suggests directions for future work.

Chapter 2

The Cooperative Cleaners Case Study

Swarm intelligence is inspired by the collective behaviour of decentralised and self organised systems found in nature, where groups of organisms seem to have some fundamentally distinct behaviours that are not present when observing an individual member of that group. The application of swarm principles to robots is called swarm robotics.

Swarm robotics [10, 11] therefore concerns systems in which a group of similar *robots*, each of which is relatively simple in its behavioural repertoire, is coordinated in a way that leads to useful behaviour of the swarm itself. Certain structural aspects of swarms are commonly assumed when swarm intelligence is discussed: apart from the *simplicity* of the constituent robots, it is also expected that a swarm has no central controller or *master* robot that leads the activities of others. Each robot is *independent* and interacts with all or part of its fellow swarm-members (as well as with other aspects of its environment) in simple ways.

Dias and Stentz [17] summarised several advantages of using swarm robots.

- Swam robots inherently enjoy the benefits of parallelism. In certain domains, a single robot may simply be unable to accomplish a given task on its own (e.g., an extremely complex and heavy task). With task decomposable application domains, robot teams can accomplish a given task more quickly than a single robot by dividing the task into subtasks and executing them concurrently. However, when designing such systems it should be noticed that simply increasing the number of robots assigned to a task does not necessarily improve the system's performance. Multiple robots must cooperate and synchronise intelligently to avoid disturbing each other's activity and achieve efficiency.

- Decentralised systems tend to be, by their very nature, much more robust than centralised systems (or systems comprised of a single complex unit). In general, a team of robots may provide a more robust solution by introducing redundancy and by eliminating any single point of failure. For example, when systems rely on a centralised controller, damage to the controller will clearly be disastrous. When using a multi-robot system, even if a large number of the robots stop working for some reason, the entire group will often be able to complete its task. For example, in space exploration, critical systems and so forth, the benefit of redundancy and robustness offered by a multi-robots system is highly noticeable.

- Another advantage of decentralised systems is the ability to dynamically reallocate subtasks between the swarm's units, thus adapting to unexpected changes in the environment, as well as scalability when the tasks assigned to a group of robots become more and more complex.

The Cooperative Cleaners case study is an example of swarm robotics. In this case study, a group of robots cooperate in order to reach the common goal of cleaning a dirty floor. A cleaning robot has a bounded amount of memory; it can only observe the state of the floor in its immediate surroundings and decide on its movement based on these observations. Consequently, the overall topology of the floor contamination is unknown to the robots. The robots use an indirect means of communication based on signals and sensing, and the desired goal of cleaning the whole floor is thus an emerging property of multi-robot cooperation.

In this chapter we specify this problem and in further chapters we will first apply the object-oriented modelling language ABS to formally model the case study and second analyse the behaviour of the model using simulations.

2.1 Definition of the Problem

In this thesis we consider the static version of *the Cooperative Cleaners problem* as given in [31]; in this version the dirty shape of the floor does not grow due to spreading of contamination. We further consider some adaptations due to ambiguities in the original description of the problem.

In the rest of this section we substantiate *the Cooperative Cleaners problem*. We will first present the dirty floor, then the cleaning robots and finally the cleaning protocol CLEAN.

2.1.1 The Floor

The shape of the floor is a region in $\mathbb{Z}^2$ which we represent as an undirected graph G. Let V be the set of vertices in G. Each element in V is a *tile* of the floor and is represented as a pair $v = (x, y)$. This pair (x, y) describes the position of v in the floor. Let E be the set of edges in G, each edge is a pair of vertices (v, w) such that v and w are connected through a *4-neighbours* relation (explained below). The dirty floor F_t is a subgraph of G, where t represent the time. In the initial state we assume that G is a *single connected component* (explained below) without holes or obstacles, and $F_0 = G$. Let us now introduce some concepts:

Four neighbours: Let V_t be the set of vertices of the dirty floor F_t. If $v \in V_t$, the *4-neighbours* of v is a set of vertices which cardinality ≤ 4, such that for all $w \in$ *4-neighbours* of v; $w \in V_t$ and w is located at the top, right, bottom or left of v.

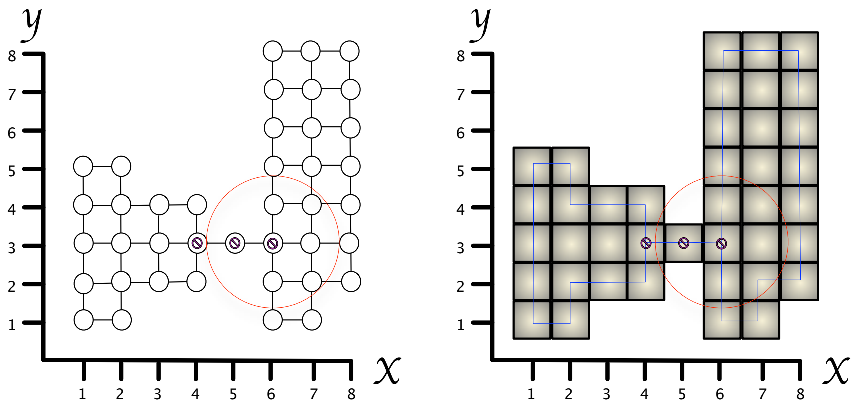


Figure 2.1: An example of a graph F_0 (to the left) representing a dirty floor (to the right)

Eight neighbours: Let V_t be the set of vertices of the dirty floor F_t. If $v \in V_t$, the *8-neighbours* of v is a set of vertices which cardinality ≤ 8 such that for all $w \in$ *8-neighbours* of v; $w \in V_t$ and w is located around v.

Adjacent: Let v and w be vertices of the dirty floor F_t. A vertex w is *adjacent* to v if and only if $w \in$ *4-neighbours* of v.

Path: Let $v_1, \ldots, v_n$ be vertices of the dirty floor F_t. A *path* in F_t is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that for $1 \leq i < n$, v_i and v_{i+1} are *adjacent*.

Vertex Connectivity: Let v and w be vertices of the dirty floor F_t. A vertex v is *connected* to w if there is a *path* from v to w.

Graph Connectivity: The dirty floor F_t is a *single connected component* if there is a *path* from every vertex to every other vertex in F_t.

Graph Boundary: Let V_t be the set of vertices of the dirty floor F_t. The *boundary* of F_t is the set of all vertices $v \in V_t$ such that for all $v \in$ *boundary* of F_t, the cardinality of the *8-neighbours* of $v < 8$.

Vertex Criticality: Let v, w and w' be vertices from the dirty floor F_t. The vertex v is *critical* if there are two different vertices w and w' such that $\{w, w'\} \subseteq$ *4-neighbours* of v, and there is no *path* from w to w' that contains only elements of the *8-neighbours* of v. Observe that if v is *critical*, then $V_t \backslash \{v\}$ does not form a *single connected component* graph.

Example 1.

Figure 2.1 (left) depicts a *single connected component* graph F_0 where:

$$\begin{aligned} V_0 = \{&(1,1),(2,1),(6,1),(7,1),(1,2),(2,2),(3,2),(4,2),(6,2),(7,2),(8,2),\\ &(1,3),(2,3),(3,3),(4,3),(5,3),(6,3),(7,3),(8,3),(1,4),(2,4),(3,4),\\ &(4,4),(6,4),(7,4),(8,4),(1,5),(2,5),(6,5),(7,5),(8,5),(6,6),(7,6),\\ &(8,6),(6,7),(7,7),(8,7),(6,8),(7,8),(8,8)\} \end{aligned}$$

4-neighbours of $(6,3) = \{(6,4),(7,3),(6,2),(5,3)\}$

8-neighbours of $(6,3) = \{(5,3),(6,4),(7,4),(7,3),(7,2),(6,2)\}$

$(4,3),(5,3)$ and $(6,3)$ are *critical* vertices

$$\begin{aligned} \textit{boundary} \text{ of } F_0 = \{&(1,1),(2,1),(6,1),(7,1),(1,2),(2,2),(3,2),(4,2),(6,2),\\ &(7,2),(8,2),(1,3),(4,3),(5,3),(6,3),(8,3),(1,4),(2,4),\\ &(3,4),(4,4),(6,4),(8,4),(1,5),(2,5),(6,5),(8,5),(6,6),\\ &(8,6),(6,7),(8,7),(6,8),(7,8),(8,8)\} \end{aligned}$$

2.1.2 The Cleaning Robots

Let us define a group of k identical cleaning robots. Each cleaning robot can move across the dirty floor F_t (moving from a vertex v to a vertex w such that $w \in$ *4-neighbours* of v). All the cleaning robots are placed at time $t = 0$ in the same initial vertex, located on the *boundary* of the initial dirty floor F_0. Each cleaning robot is equipped with a sensor capable of observing its local environment. The local environment includes the dirty vertices in an area of diameter 5 as well as other cleaning robots located in that area (as depicted in Figure 2.2). All the cleaning robots agree on a common protocol which basically consist on the robots moving and cleaning vertices (if the vertices are not *critical*) along the boundary of F_t (as depicted in Figure 2.2).

We assume that the initial dirty floor F_0 is a *single connected component* with no holes or obstacles. As an invariant, the protocol must preserve this property over time. Each vertex may contain any number of cleaning robots simultaneously. The cleaning robots do not have any prior knowledge of the shape or size of the dirty floor F_0 except that it is a *single connected component* with no holes or obstacles. The robots' goal is to clean the dirty floor (by progressively removing vertices from V_t), meaning that the robots must ensure that:

$$\exists t_{success} \text{ such that } V_{t_{success}} = \emptyset$$

The protocol must assume that all the cleaning robots are identical and that there is no explicit communication between the robots (only broadcast and sensing actions in the local environment are allowed). All the cleaning robots start and finish their task in the same vertex (as a consequence the initial vertex is artificially marked as *critical*). In addition the

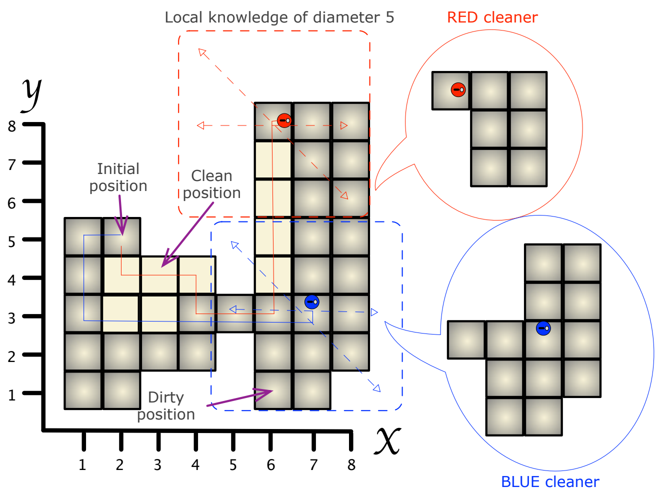


Figure 2.2: An example with two cleaning robots (named BLUE and RED) at discrete time 11

whole system should support fault-tolerance: Even if almost all the robots cease to work before completion of the mission, the remaining ones will eventually complete the mission (e.g., by ignoring the presence of non-working robots in a vertex).

2.1.3 The Cleaning Protocol

In our work with *the Cooperative Cleaners problem* we are going to use the following cleaning protocol, CLEAN [31]. This protocol is a cyclic algorithm where in each discrete time step, each cleaning robot cleans its current position (where every position is a vertex) assuming it is not critical, and moves according to a local movement rule, creating the effect of a clockwise traversal along the boundary of the dirty floor F_t. As a result, the cleaning robots "peel" layers from the boundary of F_t, until F_t is cleaned entirely (meaning that $V_t = \emptyset$). This protocol preserves the connectivity of the dirty floor F_t by preventing the cleaning robots from cleaning critical vertices. This ensures that the robots stop only upon completing their mission. An example of the running protocol can be seen in Figure 2.2, where two cleaning robots RED and BLUE are following the cleaning protocol, (shown at discrete time step 11). In Figure 2.2, the darker positions depict dirty positions, while the lighter positions depict positions that have been cleaned, the red and and blue solid lines depict the paths of the RED and BLUE robots, respectively. The dashed squares depict the local neighbourhood of the cleaning robots, and a cleaning robot located closer to a specific side in a position depicts that the cleaning robot is signalling its intended next position to be the position adjacent to that side. Note that at time 0 all the vertices were dirty as shown in Figure 2.1. A high level description of each subroutine of the CLEAN protocol is presented in Table 2.1 and the original CLEAN protocol can be found in Appendix A.

Check task completed	Detect cases where all the positions have been cleaned, therefore the cleaning robot can stop (since the goal has been reached).
Check task near completion	Identify scenarios in which there are too many cleaning robots blocking each other, preventing the cleaning of the last few positions.
Calculate next position	The next position is the *rightmost neighbour* of the current position. The *rightmost neighbour* of the current position (x, y), is calculated in the following way: starting from the previous position, scan the *4-neighbours* of (x, y) in clockwise order until another boundary position is found.
Signal next position	Signal the desired next position to the local neighbourhood.
Check if resting	Avoid groups of cleaning robots moving together as a cluster around F_t by introducing delays. These delays are introduced by checking if a cleaning robot should give priority to other cleaning robots located in the same position and signalling to the same next position. See Chapter 4 for more details.
Check if waiting	When a cleaning robot is not resting, it may synchronise with other cleaning robots in its local neighbourhood before cleaning and moving to the next position in order to avoid simultaneous cleaning and moving, which may damage the connectivity of the dirty floor F_t. See Chapter 4 for more details.
May clean current position	Clean current position if it is not *critical*.
Move to next position	Move to the next signalled position.

Table 2.1: The subtasks of CLEAN protocol

Chapter 3

The ABS Modelling Language

The Abstract Behavioural Specification language (ABS) is a modelling language for the development of executable object-oriented models. It allows abstract, yet executable, specification of systems including concurrent workflows, synchronisation between objects, and updates to the local state of objects. ABS can be situated between design-oriented and implementation-oriented specification languages. Compared to design-oriented languages like UML diagrams, ABS is fully executable and models data-sensitive control flow. Compared to object-oriented programming languages, ABS abstracts from low-level imperative data structures by means of a functional language. In addition, ABS supports variability modelling for product line engineering [14, 28] through delta-oriented specifications and feature models as well as modelling for deployment variability [3, 20, 22–24]. ABS does not support class inheritance and method overloading, instead code reuse is achieved by applying delta-oriented programming techniques [13, 28].

The kernel of the language, called Core ABS [20], targets the formal modelling of distributed systems. Core ABS uses asynchronous communication and is based on concurrent object groups (COGs) [16, 20, 21, 29], akin to Actors [1] and Erlang processes [8]. The formal syntax and semantics of Core ABS is explained in [20]. Real-Time ABS [9] extends Core ABS models with explicit and implicit time-dependent behaviour. This chapter gives an introduction to Core ABS [14, 20] and its extension with real-time [9].

3.1 Core ABS

Core ABS consists of a functional layer and a concurrent imperative layer which helps to combine functional and imperative programming styles to develop high-level executable models. In accordance to the formal semantics of Core ABS [20], COGs execute in parallel and communicate through asynchronous method calls, while internal computation inside methods may be captured using the functional language and user-defined algebraic data types.

Syntactic categories.	*Definitions.*		
T in Ground Type	T	::=	$B \mid I \mid D \mid D\langle\overline{T}\rangle$
B in Basic Type	B	::=	$\mathsf{Bool} \mid \mathsf{Int} \mid \ldots$
A in Type	A	::=	$N \mid T \mid N\langle\overline{A}\rangle$
x in Variable	Dd	::=	$\mathbf{data}\ D[\langle\overline{A}\rangle] = [\overline{Cons}];$
e in Expression	$Cons$	::=	$Co[(\overline{A\ fn})]$
b in Bool Expression	F	::=	$\mathbf{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A\ x}) = e;$
v in Value	e	::=	$b \mid x \mid v \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \mathbf{case}\ e\ \{\overline{br}\} \mid$
br in Branch			$\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \mathbf{let}\ (T\ x)\ = e\ \mathbf{in}\ e$
p in Pattern	v	::=	$Co[(\overline{v})] \mid \mathbf{null}$
	br	::=	$p \Rightarrow e;$
	p	::=	$\_ \mid x \mid v \mid Co[(\overline{p})]$

Figure 3.1: Syntax for the functional level of Core ABS. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over the corresponding syntactic categories, and square brackets [ ] optional elements.

3.1.1 The Functional Level of ABS

The functional layer of Core ABS is used to model computations on the internal data of the objects. It allows modellers to abstract from implementation details of data structures at an early stage in the software design and allow data manipulation without committing to a low-level implementation. The functional layer includes a library with some basic datatypes such as the empty type `Unit`, numbers, strings, and Boolean, with arithmetic and comparison operators, and parametric datatypes such as lists, sets, maps and functions to manipulate these datatypes using pattern matching. The library can be extended by user-defined datatypes and functions.

New datatypes with given constructors can be defined as follows:

```
data Color = PrimaryColor(String p_name) |
             MixingColor(String m_name, List<String> combinationOf);
```

Here `PrimaryColor` is a constructor and `p_name` is an accessory function. Thus `p_name(PrimaryColor("BLUE")) = "BLUE"`.

Functions map values of one type to values of another type. Function definitions in ABS use pattern matching. Below is an example of how to write functions in Core ABS:

```
def Bool isPrimary(Color c) = case c {
  PrimaryColor(_) => True;
  MixingColor(_,_) => False;
};

def String name(Color c) = case c {
  PrimaryColor(_) => p_name(c;)
  MixingColor(_,_) => m_name(c);
};
```

Formal syntax. The formal syntax of the functional language is given in Figure 3.1. The ground types T consist of basic datatypes B, names D for datatypes in the library as well as user-defined datatypes and I for interfaces. In general, a type A may also contain type variables N (i.e., uninterpreted type names [27]). In *datatype declarations Dd*, a datatype D has a set of constructors *Cons*, each of which has a name *Co* and a list of types $\overline{A}$ with accessory functions $\overline{fn}$ for their arguments. *Function declarations F* have a return type A, a function name *fn*, a list of parameters $\overline{x}$ of types $\overline{A}$, and a function body e. Both datatypes and functions may be polymorphic and have a bracketed list of type parameters (e.g., `List<Int>`).

Expressions e include variables x, Booleans b, values v, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, if-then-else expressions **`if`** e **`then`** e **`else`** e, case expressions **`case`** $e\ \{\overline{br}\}$ and let expressions **`let`** $(T\ x)\ = e$ **`in`** e. *Values v* are expressions which have reached a normal form; i.e., constructors applied to values $Co(\overline{v})$ or **`null`** (omitted from Figure 3.1 are values of basic types). *Case expressions* have a list of branches $p \Rightarrow e$, where p is a pattern. Patterns include wild cards _, variables x, values v, and constructor patterns $Co(\overline{p})$. The branches are evaluated in the listed order and fresh variables in `p` are bound in the expression `e`.

3.1.2 The Concurrent Object Level of ABS

The imperative layer of Core ABS allows a modeller to express cooperation between concurrent objects through communication and synchronisation. This subsection first explains the concurrency model of Core ABS which includes asynchronous communication and concurrent object groups (COGs), and second explains the formal syntax of the language.

Asynchronous method calls are one of the ways to model (and later implement) communication between objects [21]. With asynchronous method calls, it is always possible to send a method invocation; in this case the caller may be resynchronised with the callee after the call finishes its execution; this asynchronous way of communication is similar to the Actor model [1]. Furthermore, this way of communication can be combined with cooperative non-preemptive scheduling (in the form of *processor release points* in ABS) where objects decide when to release the processor while executing a method; compared to other concurrent object models, this reduces the waiting for replies inside objects by allowing different activities to be interleaved inside objects and it expresses the overall workflow and communication between objects in a natural manner and closer to real systems.

Concurrent object groups is a two tiered concurrency model where the upper tier is based on the Actors model [1] with asynchronous communication [21] and no shared state; while the lower tier is based in cooperative multitasking with synchronous and asynchronous calls and shared state. COGs allow the combination of active and reactive behaviour by means of cooperative multithreading. Using COGs a program's components are represented as objects or as groups of objects, and the behaviour of the overall system results from the collaboration and interaction between the COGs.

In the COGs model, the activation of a method results in a process to be executed by the called object; additionally each COG executes on its own virtual processor. Inside a COG only one object executes at a time and concurrency is expressed as interleaving, while true concurrency appears when multiple processes are executed simultaneously in different

COGs.

To have a better understanding of the two tiered concurrency model, let us consider two objects *A* and *B* located in the same COG and two objects *C* and *D* located in different COGs. When an object *A* calls a method *mb* of object *B*, it sends an invocation message to *B* including the arguments to the method. When method *mb* finishes to execute on the same processor, it sends a reply to *A* with the return values from the method activation. Object *A* may or may not continue executing while it is waiting for *B*'s reply, depending on whether the call is synchronous or asynchronous, so the cooperative multithread scheduler decides the order in which the processes are executed. On the other hand, when an object *C* calls a method *md* of object *D*, it sends an asynchronous invocation message to *D* along with the method's arguments. Method *md* executes on *D*'s processor and sends a reply to *C* when it is finished, with the return values. Object *C* may continue executing in parallel on its own processor while it is waiting for *D*'s reply. Compared to standard method invocation in a pure multithreaded setting with preemptive scheduling (discussed in Chapter 1), the approach with cooperative multithreading with nonpreemptive scheduling leads to increased parallelism in distributed models.

Below is an example of a trivial class for a server written in Core ABS.

```
interface Server {
  Bool request(Int tasksize);}

class Server implements Server {
  Bool request(Int tasksize){
    while (tasksize > 0) {
      //execute task
      tasksize = tasksize - 1;}
    return True;} }
```

This example shows the implementation of a class as a template for passive objects without a `run` method. In contrast the following example describes a class for *active* objects. The objects of this class execute their `run` method after they have been created. In the context of this example, the class `Client` is a model of an environment for the class `Server`:

```
class Client (Server s1, Server s2, List<Int> tasks) {
  Bool flag = False;

 Unit run(){
   Int task = 0;
   Bool reply = False;
   if (tasks != Nil) {
      task = head(tasks);
      tasks = tail(tasks);
      if (flag) {
        Fut<Bool> r = s1!request(task);
        reply = r.get;}
      else {
        Fut<Bool> r = s2!request(task);
        await r?;
        reply = r.get; }
      flag = ~ flag;
      this!run();} } }
```

Syntactic categories.	*Definitions.*		
C, I, m in Name	P	$::=$	$\overline{Dd}\ \overline{F}\ \overline{IF}\ \overline{CL}\ \{\ [\overline{T\,x};]\ s\}$
g in Guard	IF	$::=$	$\textbf{interface}\ I\ \{\ [\overline{Sg}]\ \}$
s in Stmt	CL	$::=$	$\textbf{class}\ C\ [(\overline{T\ x})]\ [\textbf{implements}\ \overline{I}]\ \{\ [\overline{T\ x};]\ \overline{M}\}$
	Sg	$::=$	$T\ m\ ([\overline{T\ x}])$
	M	$::=$	$Sg\ \{[\overline{T\ x};]\ s\ \}$
	g	$::=$	$b \mid x? \mid g \wedge g$
	s	$::=$	$s;s \mid \textbf{skip} \mid \textbf{if}\ b\ \{\ s\ \}\ [\textbf{else}\ \{\ s\ \}] \mid \textbf{suspend} \mid$
			$\textbf{while}\ b\ \{\ s\ \} \mid \textbf{return}\ e \mid x = rhs \mid \textbf{await}\ g$
	rhs	$::=$	$e \mid \textbf{new}\ [\textbf{cog}]\ C\ (\overline{e}) \mid e.\textbf{get} \mid o!m(\overline{e}) \mid o.m(\overline{e})$

Figure 3.2: Syntax for the concurrent object level of Core ABS.

Here we see how the functional layer is combined with the imperative layer as `List<Int>` is a functional data structure with functions `head` and `tail`.

In general classes in Core ABS may express both active and reactive behaviour. The main block describes the initial configuration of a model, below is an example of a main block for the client-server example:

```
{
  Server server1 = new cog Server();
  Server server2 = new cog Server();

  new cog Client(server1,server2,list[2,4,6,8,6,4,2]);
  new cog Client(server1,server2,list[1,3,5,7,5,3,1]);
}
```

As it is possible to observe through the example, methods and main blocks may be implemented by some variable declarations and a sequence of statements. Core ABS enforces variable declarations to be initialised. Also each method has a signature that coincides with the signature in the corresponding interface.

Formal syntax. The formal syntax of the imperative layer is given in Figure 3.2. The basic structure of a system P in Core ABS includes a set of data definitions $\overline{Dd}$, a set of functions $\overline{F}$, a set of interfaces $\overline{IF}$, a set of classes $\overline{CL}$ and a main block $\{\ [\overline{T\,x};]\ s\}$, which is similar to a method body.

An interface IF has a name I and method signatures Sg. A method signature Sg has a return type T, a name m and an optional set of parameters $\overline{T\,x}$. A class CL has a name C, interfaces $\overline{I}$ (specifying types for its instances), class parameters and state variables x of type T, and methods M. The *attributes* of the class are both its parameters and state variables. Classes may have an optional initialisation block used for more complex initialisation. M defines a method with signature Sg, local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement s. Statements may access attributes, locally defined variables, and the method's formal parameters.

Core ABS has a standard set of statements s including sequential composition $s;s$, assignment $x = rhs$, **`skip`**, **`if`** $-$ **`else`**, **`while`** and **`return`**. In relation to the semantics for

Syntactic categories.	*Definitions.*		
d in DurationExpr	e	$::=$	$\ldots \mid d \mid$ **now**$()$
	g	$::=$	$\ldots \mid$ **duration**(d, d)
	s	$::=$	$\ldots \mid$ **duration**(d, d)

Figure 3.3: Syntax extending Core ABS with real-time

processor release points, Core ABS has the statements **suspend** and **await**. The statement **suspend** unconditionally releases the processor and suspends the process. In **await** g, the guard g is a boolean condition; if g evaluates to `False` the processor is released and the process suspended. When the processor is free any enabled process from the suspended process queue of the COG can start execution. In addition to side-effect-free expressions and basic types coming from the functional language; there are expressions *rhs* for method invocation, object creation and two implicitly defined read-only variables: the object level variable **this** to access the reference of the current object and the process level variable **destiny** to access the return address to the process itself. Objects can be created in the same COG (by **new** $C\ (\overline{e})$) or in a fresh COG (by **new cog** $C\ (\overline{e})$).

Communication in Core ABS is based in asynchronous method calls $o!m(\overline{e})$, where o is the reference to the called object. After the asynchronous call the caller may proceed its execution. If the caller is expecting a return value ($x = o!m(\overline{e})$), it will be stored in a *future variable* x. A future variable **Fut**$\langle T\rangle\ x$ refers to a mailbox (or future) for a value which has not been computed yet; here T corresponds to the return type of the called method. The expression x.**get** is used to retrieve the value from a future referenced by x once it has been computed. If no value is stored in the future, the process is blocked until the value is computed. The statement **await** $x?$ suspends the process until the value in the future referenced by x is computed. The blocking call $y = o.m(\overline{e})$ is syntactic sugar for the following sequences of statements (assuming that x is a fresh name):

Fut$\langle T\rangle\ x = o!m(\overline{e});\ y = x.$**get**;

Similarly the non-blocking call **await** $y = o.m(\overline{e})$ is syntactic sugar for the following sequences of statements:

Fut$\langle T\rangle\ x = o!m(\overline{e});$ **await** $x?;\ y = x.$**get**;

3.2 Real-Time ABS

Figure 3.3 shows an extension of the the syntax of Core ABS for the explicit and implicit modelling of time [9]. With explicit time, the modeller inserts duration statements with best and worst-case execution times into the model. This is the standard approach to modelling timed behaviour, well-known from, e.g., timed automata in UPPAAL [26]. Duration statements specify explicit execution times. A process that pauses for some amount of time is explicitly modelled by means of the statement **await duration**(min, max). In this case, the process is suspended for an amount of time between min and max, where min and max are values of type Duration. In contrast, a computation in an object which lasts for a certain amount of time is specified by the statement **duration**(min, max), in this case the process blocks for an amount of time between *min* and *max* and no other process can be scheduled.

With implicit time the execution time is not specified explicitly in terms of durations, but rather *observed* on the executing model. The function **`now()`** returns the current global time, the return value of **`now()`** can be used to, e.g., compare time values or monitor a system's response time and then observe the implicit range of time. With implicit time, the modeller does not make local assumptions about the execution time of a model. The execution time depends on how quickly the global system can advance and how this imposes local constrains where the observations occur.

3.3 Other Extensions of Core ABS

Real-Time ABS with *deployment components* [3,20,22–24] extends Core ABS with an approach to integrating deployment architectures and resource consumption into models written in Core ABS. The approach is based on a separation of concerns between the resource cost of performing computations and the resource capacity of the deployment architecture. A deployment component captures the execution capacity of a location in the deployment architecture, on which a number of concurrent objects are deployed. The capacity is specified as an amount of resources which is available per time interval. The separation of concerns between cost and capacity allows the performance of a model to be easily compared for a range of deployment choices.

Core ABS is also extended with *variability modelling* using delta-oriented framework [12,14]; variability is used to specify that multiple similar models can be created by selecting different instances of features and applying them to a variable source model. In this variability framework, given a set F of features for a model, a common base model P, a set of delta modifications Δ where the sets of features from the feature model F are linked to sets of delta modifications from the delta model Δ, it is possible to produce different product line configurations $C_1, C_2, \ldots, C_n$, and from any C_i it is possible to extract different specific products.

3.4 Tool Support

The ABS language has been implemented on different backends, in particular for Java and Maude [15]. A full description of the ABS tool suite can be found in [33]. The Java backend for ABS creates Java code that is compiled and executed on the JVM (Java Virtual Machine). The formal semantics of ABS is defined in rewriting logic, which is executable on the Maude tool [15]. A code generator creates Maude terms corresponding to ABS classes and functions, which can be executed by directly using the formal semantics of ABS [20] as a language interpreter. Code editing and compilation support is provided for both the Eclipse IDE and Emacs editor. Finally, COSTABS [2] is a static cost analyser for cost and termination for models written in Core ABS. The COSTABS tool provides worst-case estimations for e.g., resource consumption, the number of tasks that are spawned along the execution, the number of objects created along the execution and the approximate number of executed instructions.

Chapter 4

The Cooperative Cleaners: Model in Real-Time ABS

This chapter describes how the model of *the Cooperative Cleaners problem* introduced in Chapter 2; can be specified in Real-Time ABS. Our model is based on the original algorithm presented in [31] (see Appendix A). As depicted in Figure 4.1, this model can be summarised as the interaction between the environment and the cleaning robots.

Following the general structure of a model in ABS, we first define user-defined datatypes to represent the information stored in the environment and the local information stored in the cleaner's limited memory, second we explain how we implement the case study with two classes representing the environment and the cleaners, respectively and third we explain how we setup the main block. Different initial configurations can be expressed in the main block, which varies in, e.g., the shapes of the initial dirty floor, the number of cleaning robots, the initial position, etc.

The *environment* in our model contains all the information that any of the cleaners can observe and sense; e.g., the floor, the position of other cleaners on the floor, the signalling that other cleaners send to their neighbourhood, the status of other cleaners and so forth. The *local environment* of a cleaner is the part of the environment that the cleaner can observe and sense (which is represented in our model as a query from a cleaner to the environment). The local environment includes the floor, the position of other cleaners in the floor, etc., but

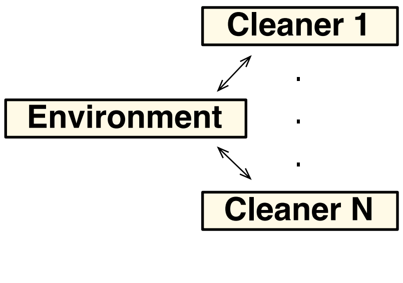


Figure 4.1: 1..N cleaners interacting with the environment

in a diameter of size 5 as depicted in Figure 2.2. Note that according to the definition of the problem described in Chapter 2, the cleaners do not communicate directly with each other, but instead they sense and observe their local environment in order to decide what to do according to a protocol.

As shown in Figure 4.2, in order to run a simulation in our model we will first create an environment which contains a dirty floor where the shape of the floor is assumed to be a single connected component with no holes. Second we create a finite number of cleaning robots. Each cleaning robot gives an initial signal to the environment indicating its initial position which is assumed to be on the boundary of the dirty floor, and starts the CLEAN protocol. Each cleaner runs one iteration of the CLEAN protocol per time interval. The protocol is run as long as necessary until the common goal of cleaning the dirty floor is reached; after that each cleaner *stops*.

In our model we have made the decision to model parts of the internal computation (of the methods in the classes) in the functional layer of ABS, which makes the imperative part more concise, hopefully easier to read, and more focus in the control flow and communication. The model contains a considerable number of functions. In this chapter we are going to present and explain a subset of the functions. For the complete specification code of the cooperative cleaners, see Appendix C.

The rest of this chapter explains our ABS model. Section 4.1 summarises the modelling decisions and challenges we encountered while building our model. Sections 4.2, 4.3 and 4.4 explain in detail the environment, the cleaners and the initial configuration, respectively.

4.1 Challenges of Modelling the Cooperative Cleaners in Real-Time ABS

The model presented in this chapter has been accomplished after a series of iterations where we first polished the abstractions and the user-defined datatypes in order to achieve simplicity and clarity, and later understood and corrected some of the errors found while running a number of simulations varying the shape of the dirty floor, the number of cleaners and the initial position. We also needed to guess, interpret and adapt many of the ambiguous and underspecified definitions found in the original protocol description [31]. Our initial aim was to make a precise implementation of the algorithm in order to have an executable model that can verify the claims of the original paper, but while building the model we needed to interpret ambiguities according to our understanding, and adapt the protocol according to the general principles of swarm robotics and according to the object-oriented and timed framework of Real-Time ABS. As a result we do not have an exact implementation of the original protocol, but it is as close as we managed to reproduce.

In the rest of this section we are going to summarise the challenges we encountered and later in the chapter we will explain how these challenges were overcome while building our model of the cooperative cleaners.

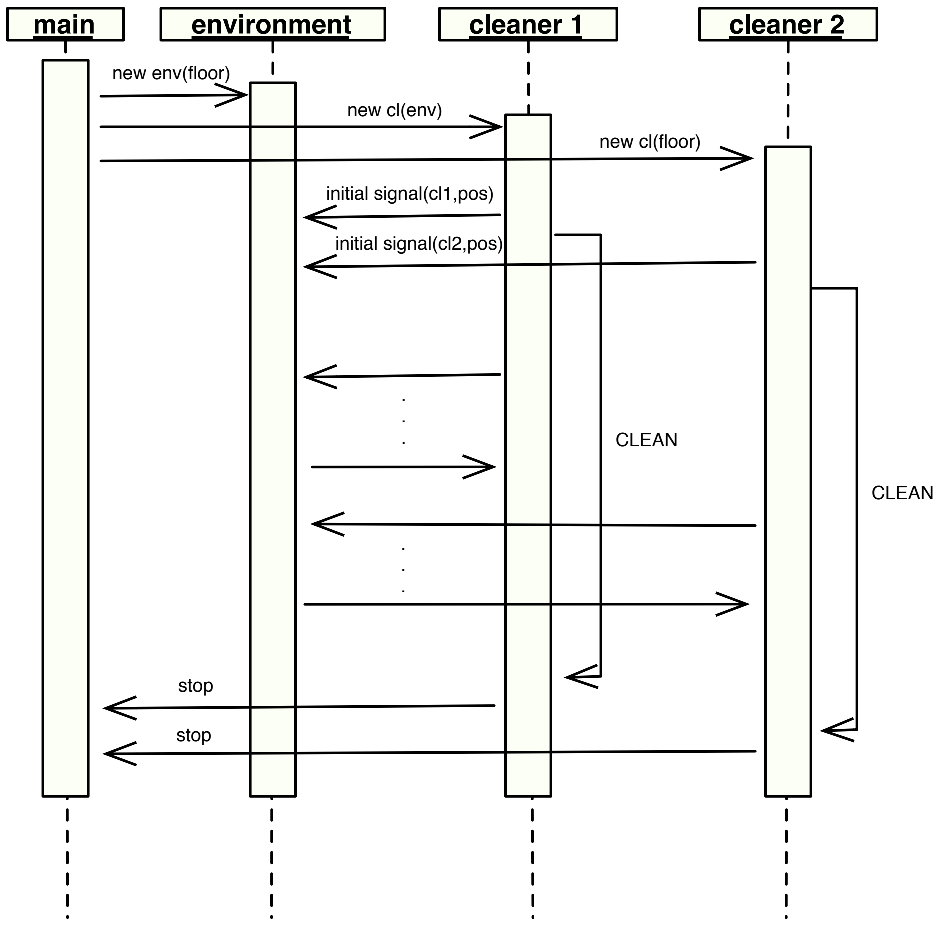


Figure 4.2: Rough sequence diagram of an initial configuration with two cleaners

4.1.1 The User-defined Datatypes and Abstractions.

As presented in Chapter 2, there are two entities in the case study: the floor and the cleaning robots. The floor is a single connected component graph that is divided into tiles (positions) and has a dirty area which is progressively cleaned over time. The cleaning robots have limited memory to store their current state, hardware for sensing the floor and other cleaning robots, hardware for signalling to other robots, hardware for cleaning tiles and hardware for moving around the floor. Moreover as it can be understood in the definition of the problem, the floor is a passive entity while the cleaning robots are the ones who possess intelligence as a swarm.

Our primary interest in the case study was to focus on the concurrent behaviour of the cleaning robots, the implicit interaction between them and in the correctness of the CLEAN protocol. For this reason we decided to abstract the modelling of the hardware of the cleaning robots and instead use method calls. Our solution was to create an *environment* entity which as explained before acts as a sort of server (from which is reactive in contrast to the passive floor) and that the cleaning robots use as an abstraction of the behaviour of their hardware.

It was also part of our modelling decision how to represent the information of the floor and the cleaning robots as user-defined datatypes and how to manipulate this information. Both the user-defined datatypes and their associated functions where used by the environment as well as by the cleaners. We also decided what data was interesting to record as monitoring information in the environment and in the cleaners, and how to represent it as user-defined datatypes.

4.1.2 The Interpretation and Representation of Concepts as Functions.

As presented in Chapter 2, in the original paper the authors give an informal description of the CLEAN protocol which includes informal concepts referring to the floor e.g., *4-neighbours*, *8-neighbours*, *single connected component*, *boundary* of a graph, *critical* tile, and so forth; and informal concepts referring to the algorithm itself e.g., *rightmost neighbour*, calculation of *resting dependencies*, calculation of the *waiting set* and so forth. These concepts were explained implicitly and by informal definitions but we need explicit definitions for our executable model. It was part of our work to interpret these concepts and find suitable representations of them in the functional language of ABS.

4.1.3 The Ambiguities of the CLEAN Protocol.

The original definition of the CLEAN protocol in [31] contains ambiguities and uncertainties. Below we provide a list of the places in the algorithm that we needed to give special attention in order to keep our proposed executable model explained in this chapter as faithful as possible to the original problem definition.

1. *First movement of the cleaner.* In the original definition of the CLEAN protocol, the first movement of a cleaner is determined by the charts in Figure A.2 of Appendix A. The

authors also explain that all possible configurations do not appear in those charts, but could be obtained by applying rotation and mirroring to the ones from the figure.

2. *Check Near Completion of Mission.* As it can be observed in Appendix A, the original definition of this subtask contains a condition which is literally specified as "this is the end of the tour". The meaning of this condition is not explained in the paper.

3. *Calculate waiting dependencies.* As it can be observed in Appendix A, the original algorithm seems to allow a cleaning robot to read and write on the internal state of other cleaning robots while calculating its waiting set. From our point of view this subtask violates the principles of independence in swarm robotics and encapsulation in object-orientation. Instead we aim for an autonomous and decentralised solution.

4. *Implicit Synchronisation.* As it can be observed in Figure A.1 of Appendix A, at each time step each cleaning robot performs one or more cycle of the CLEAN protocol in coordination with other cleaning robots in the local neighbourhood. This case study as it is defined in the original paper, seems to assume a sort of implicit barrier synchronisation between the cleaning robots in the local neighbourhood. Here is an example where such synchronisation is needed: each cleaning robot will decide if it will change its status to "resting" depending on the signalling of the intended next position of other cleaners placed in the same current position. For a cleaning robot to decide if it should be resting or not, it will assume that the other cleaning robots in its local neighbourhood have already signal their intended next position. A similar situation occurs when a cleaning robot begins to calculate its waiting set.

5. *Time Model.* The original definition of the problem in [31] makes reference to a time model for which the semantics is not explained in the paper. In Appendix A, it is possible to observe that steps 5, 6 and 9 in the CLEAN protocol depend on a "current time step" but is not explained in the protocol how and when time advances and what is meant by the current time step.

It was part of our modelling decision how we interpreted and adapted these ambiguities. See Section 4.3.3 for further explanation.

4.2 The Environment

The *environment* in our model encapsulates the shared information available to the different cleaning robots. The shared information includes the current shape of the dirty floor, and relevant information concerning the different cleaners, e.g., their position, their next intended position, etc. In this section we will first explain the user-defined datatypes we use for representing the floor and the information that the cleaners can sense and observe locally in the environment, and later we will explain how the environment is modelled in the class `EnvironmentImp` with interface `Environment`. We will also explain the main functions used for manipulating the datatypes.

4.2.1 User-defined Datatypes

The environment uses datatypes to represent and manipulate information about the floor and the cleaners, and information for monitoring the model during the analysis explained in Chapter 5.

The Floor. The user-defined datatype `Graph` represents the floor (defined in Section 2.1). The floor and the dirty subgraph of the floor are modelled in ABS as sets of vertices. Each vertex is represented as a position. The vertices `Pos` are represented in ABS as pairs (x, y). The datatype `PosSet` represents a set of positions and will be used for calculating the different definitions from Chapter 2. The constructors `EmptySet` and `Insert(v,s)` represent empty and non-empty sets, respectively. Here `v` is a value to be added to the set `s`.

```
type Pos = Pair<Int, Int>;
type Graph = Set<Pos>;
type PosSet = Set<Pos>;
```

The Cleaners. The following datatypes are used to represent information about the cleaners. According to the problem definition given in Chapter 2, more than one cleaner can be in the same position, for that reason the datatype `CleanersPerPos` is a map from positions `Pos` to another map of cleaners `Cleaners`. The map `Cleaners` maps a cleaners' name of type `String` to triples of the form `Triple<Status,Time,Int>`. These triples represent the status of a cleaner, the time in which the cleaner moved to its current position and the priority calculated according to Table 2.1, respectively. Following the CLEAN algorithm described in Section 2.1.3, a cleaner can be in three different states; `Active`, `Resting` or `Stop`. Finally the datatype `CleanerSignal` is a map from the cleaners' names to their intended next position. The constructors `EmptyMap` and `InsertAssoc(Pair(k,v),m)` represent empty and non-empty maps, respectively. Here `Pair(k,v)` is a pair consisting of a key `k` and its associated value `v` to be added to the map `m`.

```
type CleanersPerPos = Map <Pos,Cleaners>;
type Cleaners = Map<String,Triple<Status,Time,Int>>;
data Status = Resting | Active | Stop;
type CleanerSignal= Map<String, Pos>;
```

Monitoring the Environment. The datatypes `FloorProgress` and `FloorPath` are lists used for recording information about the environment over time.

The list `FloorProgress` records how the size of the floor is decreasing over time where the pair `Pair<Time,Int>` represents a time value and the size of the floor at that time. The list `FloorPath` records how the floor has been cleaned. The triple `Triple<Time,Pos,String>` represents the time when a position has been cleaned and and the cleaner's name. Constructors `Nil` and `Cons(v,l)` represent empty and non-empty lists, respectively. Here `v` is a value to be added in front of the list `l`.

```
type FloorProgress = List<Pair<Time,Int>>;
type FloorPath = List<Triple<Time,Pos,String>>;
```

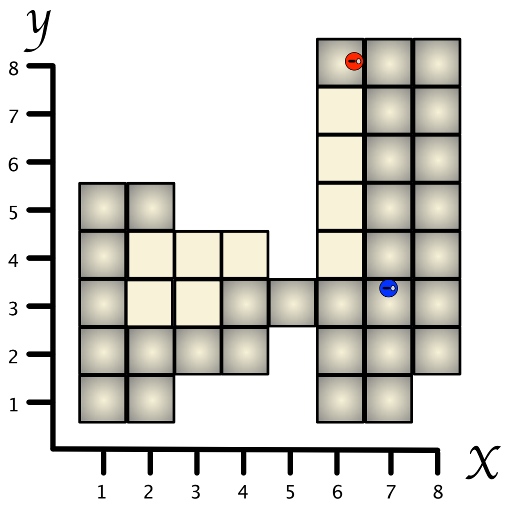


Figure 4.3: An example with two cleaning robots from the environment perspective

Example 2.

Figure 4.3 depicts a snapshot of two cleaners (RED and BLUE) while cleaning a dirty floor at the time step 11.

The *dirty sub-graph* F_t is as follows:
$F_{11} = \{(1,1),(2,1),(6,1),(7,1),(1,2),(2,2),(3,2),(4,2),(6,2),(7,2),(8,2),$
$(1,3),(4,3),(5,3),(6,3),(7,3),(8,3),(1,4),(7,4),(8,4),(1,5),(2,5),$
$(7,5),(8,5),(7,6),(8,6),(7,7),(8,7),(6,8),(7,8),(8,8)\}$

The *initial position* of RED and BLUE is $(2,5)$

The *cleaners per position map* is as follows:
$[(6,8) \mapsto [\text{RED} \mapsto (Active,10,1)],\ (7,3) \mapsto [\text{BLUE} \mapsto (Active,10,2)]]$

The *cleaners' signal map* is as follows:
$[\text{RED} \mapsto (7,8),\ \text{BLUE} \mapsto (7,4)]$

The *floor progress list* is as follows:
$[(10,32),(5,37),(0,40)]$

The *floor path list* is as follows:
$[(10,(6,7),\text{RED}),(9,(6,6),\text{RED}),(8,(6,5),\text{RED}),(7,(3,3),\text{BLUE}),$
$(7,(6,4),\text{RED}),(6,(2,3),\text{BLUE}),(3,(4,4),\text{RED}),$
$(2,(3,4),\text{RED}),(1,(2,4),\text{RED})]$

4.2.2 The Implementation of the Environment

The following interface lists all the operations that the cleaning robots use while interacting with the environment:

```
interface Environment {
   // Methods for initialising the environment
   Unit initSignal(String name, Pos pos);

   // Get local environment
   Graph getLocalDirtyFloor(Pos pos);
   CleanersPerPos getLocalCleaners(Pos pos);
   CleanerSignal getLocalSignals(Set<String> cleaners);

   // Update Environment
   Unit cleanTile(Pos pos, String name);
   Unit updateDestinationSignal(String name, Pos nextPos);
   Unit updateStatus(String name, Pos pos, Status s, Time t,Int pr );
   Unit updatePos(String name, Pos pos, Pos nextPos,
                  Status s, Time t,Int pr );
   Unit stopSignalDestination(String name);
}
```

The class `EnvironmentImp` implements the interface `Environment`; the set `dirtyFloor` contains the current shape of the dirty sub-graph, `cleanersXPos` contains the map with information about the cleaners, `signalsMap` contains a map with information about the intended next position of the cleaners, and the lists `floorPath` and `floorProgress` contain monitoring information.

```
class EnvironmentImp(Graph g) implements Environment {
  Graph dirtyFloor = g;
  CleanersPerPos cleanersXPos = EmptyMap;
  CleanerSignal signalsMap = EmptyMap;

  FloorPath floorPath = Nil;
  FloorProgress floorProgress = Nil ;
...
}
```

Initial Signalling

This method is called by the cleaners to initiate contact with the environment. In this case a cleaner provides its name and its initial position. The environment registers this information in the map `cleanersXPos`.

```
Unit initSignal(String name, Pos pos) {
     Cleaners tmpMap = put(lookupDefault(cleanersXPos, pos, EmptyMap),
                 name,Triple(Active,now(), 0));
     cleanersXPos = put(cleanersXPos, pos, tmpMap);
  }
```

Here the function `lookupDefault(m, k, v)` and `put(m, k, v)` are standard operations over a map `m`. The function `lookupDefault` searches for the key `k` in the map `m` and

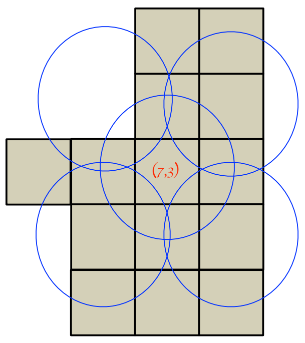


Figure 4.4: An example of how we calculate the local floor for the BLUE cleaner

returns the value associated to `k`; in case `k` is not present in the map, it returns the default value `v`. The function `put` updates the map `m` where the value associated with the key `k` is replaced by the value `v`.

Local Dirty Sub-graph

This method returns the local dirty sub-graph of diameter five around a given position. We use the function `localGraph` to calculate the local dirty subgraph of a cleaner located in a position `pos`.

```
Graph getLocalDirtyFloor(Pos pos) { return localGraph(pos, dirtyFloor); }
```

Local graph. The function `localGraph` uses various auxiliary functions explained below. The function `union` implements the normal union operator $\cup$ for sets. Figure 4.4 depicts how we are calculating the local floor for the BLUE cleaner in position $(7,3)$.

```
def Graph localGraph(Pos p, Graph g) =
    let (Graph cs) = eightNbr(p, g) in
    let (Graph uls) = eightNbr(upleft(p), g) in
    let (Graph urs) = eightNbr(upright(p), g) in
    let (Graph dls) = eightNbr(downleft(p), g) in
    let (Graph drs) = eightNbr(downright(p), g) in
    union(cs, union(uls,union(urs,union(dls,drs))));
```

Eight Neighbours. Following the definition in Section 2.1, the *8-neighbours* is modelled in ABS as follows: Given a position `p` and a dirty subgraph `g`, `eightNbr` calculates the positions `up`, `upleft`, `upright`, `left`, `right`, `down`, `downleft` and `downright`, and then filters those which are dirty.

```
def PosSet eightNbr(Pos p, Graph g) =
   let (Pos u)=up(p) in
   let (Pos ul)=upleft(p) in
   let (Pos ur)=upright(p) in
   let (Pos l)=left(p) in
   let (Pos r)=right(p) in
   let (Pos d)=down(p) in
   let (Pos dl)=downleft(p) in
```

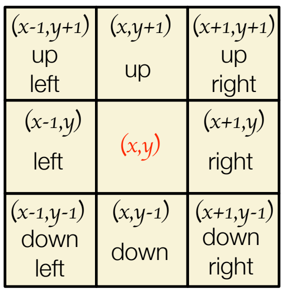


Figure 4.5: Positions around the vertex (x, y)

```
let (Pos dr)=downright(p) in
dirtySet(set[u,ul,ur,l,r,d,dl,dr],g);
```

Calculation of Different Vertices Around a Vertex. Given any position `p`, the following functions calculate the positions `up`, `upleft`, `upright`, `left`, `right`, `down`, `downleft` and `downright`, respectively as in Figure 4.5.

```
def Pos up(Pos p) = case p {Pair(x,y) => Pair(x,y+1);};
def Pos upleft(Pos p) = case p {Pair(x,y) => Pair(x-1,y+1);};
def Pos upright(Pos p) = case p {Pair(x,y) => Pair(x+1,y+1);};
def Pos left(Pos p) = case p {Pair(x,y) => Pair(x-1,y);};
def Pos right(Pos p) = case p {Pair(x,y) => Pair(x+1,y);};
def Pos down(Pos p) = case p {Pair(x,y) => Pair(x,y-1);};
def Pos downleft(Pos p) = case p {Pair(x,y) => Pair(x-1,y-1);};
def Pos downright(Pos p) = case p {Pair(x,y) => Pair(x+1,y-1);};
```

Dirty Set. Given a set of positions `s` and a dirty subgraph `g`, filter the positions from `s` that are in `g`.

```
def PosSet dirtySet(PosSet s, Graph g)=
  case s {
     EmptySet => EmptySet ;
     Insert(p,tail) =>
       case posIsDirty(p,g) {
          True => Insert(p,dirtySet(tail,g));
          False => dirtySet(tail,g);
       };
  };
```

Checking Dirtiness Status of a Vertex. Given a dirty subgraph `g` and a position `p`, if `contains(g, p)` then the position is dirty. The function `contains` implements the standard $\in$ operation for sets.

```
def Bool posIsDirty(Pos p,Graph g) = contains(g, p);
```

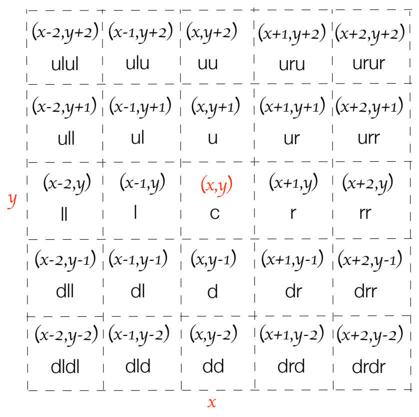


Figure 4.6: An example of how we model the sensing of the cleaners in the local neighbourhood

Map with Local Cleaners

The method `getLocalCleaners` returns a map of type `CleanersPerPos` with the cleaners located in the floor inside a diameter of size five around a position.

```
CleanersPerPos getLocalCleaners(Pos pos) {
   return localCleaners(pos,cleanersXPos); }
```

Local cleaners. The function `localCleaners` builds this map, as shown in Figure 4.6. This function uses the function `simplifyCleaners` explained below.

```
def CleanersPerPos localCleaners(Pos p,CleanersPerPos cXp ) =
   let (Pair<Pos,Cleaners> ulul)=
       Pair(upleft(upleft(p)), lookupDefault(cXp,upleft(upleft(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> ulu)=
       Pair(upleft(up(p)), lookupDefault(cXp,upleft(up(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> uu)=
       Pair(up(up(p)), lookupDefault(cXp,up(up(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> uru)=
       Pair(upright(up(p)), lookupDefault(cXp,upright(up(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> urur)=
       Pair(upright(upright(p)), lookupDefault(cXp,upright(upright(p)),
             EmptyMap)) in

   let (Pair<Pos,Cleaners> ull)=
       Pair(upleft(left(p)), lookupDefault(cXp,upleft(left(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> ul)=
       Pair(upleft(p), lookupDefault(cXp,upleft(p),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> u)=
       Pair(up(p), lookupDefault(cXp,up(p),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> ur)=
       Pair(upright(p), lookupDefault(cXp,upright(p),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> urr)=
       Pair(upright(right(p)), lookupDefault(cXp,upright(right(p)),
             EmptyMap)) in

   let (Pair<Pos,Cleaners> ll)=
       Pair(left(left(p)), lookupDefault(cXp,left(left(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> l)=
       Pair(left(p), lookupDefault(cXp,left(p),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> c)=
       Pair(p, lookupDefault(cXp,p,
             EmptyMap)) in
   let (Pair<Pos,Cleaners> r)=
       Pair(right(p), lookupDefault(cXp,right(p),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> rr)=
       Pair(right(right(p)), lookupDefault(cXp,right(right(p)),
             EmptyMap)) in

   let (Pair<Pos,Cleaners> dll)=
       Pair(downleft(left(p)), lookupDefault(cXp,downleft(left(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> dl)=
       Pair(downleft(p), lookupDefault(cXp,downleft(p),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> d)=
       Pair(down(p), lookupDefault(cXp,down(p),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> dr)=
       Pair(downright(p), lookupDefault(cXp,downright(p),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> drr)=
       Pair(downright(right(p)), lookupDefault(cXp,downright(right(p)),
             EmptyMap)) in

   let (Pair<Pos,Cleaners> dldl)=
       Pair(downleft(downleft(p)),
             lookupDefault(cXp,downleft(downleft(p)), EmptyMap)) in
   let (Pair<Pos,Cleaners> dld)=
       Pair(downleft(down(p)), lookupDefault(cXp,downleft(down(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> dd)=
       Pair(down(down(p)), lookupDefault(cXp,down(down(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> drd)=
       Pair(downright(down(p)), lookupDefault(cXp,downright(down(p)),
             EmptyMap)) in
   let (Pair<Pos,Cleaners> drdr)=
       Pair(downright(downright(p)),
             lookupDefault(cXp,downright(downright(p)), EmptyMap)) in

   simplifyCleaners(map[ulul,ulu,uu,uru,urur,ull,ul,u,ur,urr,ll,l,c,
                           r,rr,dll,dl,d,dr,drr,dldl,dld,dd,drd,drdr]);
```

Simplify cleaners. The function `simplifyCleaners` removes from a map of type `CleanersPerPos` all the positions which do not contain cleaners. The function `snd(p)` is a function over a pair `p` that returns the second element of a pair.

```
def CleanersPerPos simplifyCleaners(CleanersPerPos c) =
case c { EmptyMap => EmptyMap;
        InsertAssoc(head,tail) =>
          case snd(head)== EmptyMap {
            True => simplifyCleaners(tail);
            False => InsertAssoc(head,simplifyCleaners(tail));
          };
};
```

Map with Local Signals

The method `getLocalSignals` returns a map of type `CleanerSignal` with the cleaners included in the set `cleaners`.

```
CleanerSignal getLocalSignals(Set<String> cleaners) {
  return localSignals( cleaners, signalsMap);
}
```

Local signals. The function `localSignals` builds this map which contains the intended next positions of the cleaners in `cleaners`. The function `lookup(m,k)` returns the value associated to the key `k`, wrapped as a type `Maybe<A>`, if the key is not in the map `m` then it returns the value `Nothing`.

```
//definition of the datatype Maybe as defined in the library.
//data Maybe<A> = Nothing | Just(A fromJust);

def CleanerSignal localSignals(Set<String> cleaners,
                               CleanerSignal fullSignalsMap) =
  case cleaners {
    EmptySet => EmptyMap;
    Insert(n,tail) =>
      case (lookup(fullSignalsMap,n)== Nothing) {
        True => localSignals(tail, fullSignalsMap);
        False => InsertAssoc(Pair(n,fromJust(lookup(fullSignalsMap,n))),
                          localSignals(tail, fullSignalsMap));
      };
  };
```

Clean a Position of the Floor

The method `cleanTile` removes the position `pos` from the `dirtyFloor`. In order to guarantee that the dirty floor does not get disconnected after cleaning a position, we use an assertion with the function `isSingleConnectedComponnet(newFloor)` to check the connectivity of the dirty floor after removing `pos`. Here the function `remove(s,e)` is a function over sets which removes the element `e` from the set `s` and returns the new set without `e`.

```
Unit cleanTile(Pos pos, String name, Time timeStep) {
   Graph newFloor = remove(dirtyFloor, pos);
   assert isSingleConnectedComponnet(newFloor);
   dirtyFloor = newFloor;
   floorPath = Cons(Triple(timeStep, pos, name), floorPath);
 }
```

Observe that if during a simulation a cleaner cleans a critical position, the assertion would fail resulting in an error and stopping the execution.

Graph Connectivity. The function `isSingleConnectedComponent` returns true if the visited elements using the *depth-first search* algorithm is equal to the dirty subgraph g, in other words if all the positions can be reached from any random position. The function `emptySet` checks if a set is empty. The function `compareSet` returns true if the two sets have the same elements. The function `take` takes a random element from a set.

```
def Bool isSingleConnectedComponent(Graph g) =
    case emptySet(g) {
      True => True;
      False => compareSet(g, dfs(set[take(g)],EmptySet,g));
   };
```

Depth-first Search. This function implements the *depth-first search* algorithm in ABS. The function `dfs` returns the set of all visited positions stored in `visited`. The search starts from any random position in `dirtyFloor`. This random position will be the root in the searching tree. A position can be visited if there is a path from the initial root position to it (see Section 2.1). This function will be used to check the connectivity of a graph.

```
def Graph dfs(Graph dfsNodes, Graph visited, Graph dirtyFloor) =
   case dfsNodes {
      EmptySet => visited;
      Insert(n,newDfsNodes) => case contains(visited,n) {
         True => dfs(newDfsNodes,visited,dirtyFloor);
         False => dfs(union(newDfsNodes, fourNbr(n, dirtyFloor)),
                         insertElement(visited,n ),dirtyFloor);};};
```

Four Neighbours. Following the definition in Section 2.1, the *4-neighbours* is modelled in ABS as follows: Given a position `p` and a dirty subgraph `g`, `fourNbr` calculates the positions `up`, `down`, `left` and `right`, and then filters those which are dirty.

```
def PosSet fourNbr(Pos p, Graph g)=
  let (Pos u)=up(p) in
  let (Pos d)=down(p) in
  let (Pos l)=left(p) in
  let (Pos r)=right(p) in
  dirtySet(set[u,d,l,r],g);
```

Signal to Next Position

The method `updateDestinationSignal` is called by the cleaning robots when they want to signal their intended next position. This method updates the map `signalsMap` with the corresponding signalling information, indicating the name of the cleaning robot `name` and the intended next position `nextPos`.

In the same way the method `stopSignalDestination` removes the signalling information from the map `signalsMap`. Here the function `removeKey(m,k)` is a function over a map that removes the pair `Pair(k,v)` from the map `m` and returns a new map without this pair.

```
Unit updateDestinationSignal(String name, Pos nextPos) {
    signalsMap = put(signalsMap,name,nextPos);
}

Unit stopSignalDestination(String name) {
   signalsMap = removeKey(signalsMap,name);
}
```

Update Status

The method `updateStatus` is called by the cleaning robots when they want to change their current status. This method updates the map `cleanersXPos`.

```
Unit updateStatus(String name, Pos pos, Status s,Time t,Int pr ) {
    Cleaners tmpMap = put(lookupDefault(cleanersXPos, pos, EmptyMap),
                          name,Triple(s,t,pr));
    cleanersXPos = put(cleanersXPos, pos, tmpMap);
}
```

Update Position

The method `updatePos` is called by the cleaning robots when they want to move to their next position. This method updates the map `cleanersXPos` by first removing the cleaner from its current position in the map and second adding the cleaner to its next position.

```
Unit updatePos(String name, Pos pos, Pos nextPos, Status s,
               Time t, Int pr ) {
    Cleaners rmvTmpMap = removeKey(lookupDefault(cleanersXPos,
                         pos, EmptyMap),name);
    Cleaners addTmpMap = put(lookupDefault(cleanersXPos, nextPos,
                         EmptyMap),name,Triple(s,t,pr));
    cleanersXPos = put(cleanersXPos, pos, rmvTmpMap);
    cleanersXPos = simplyfyCleaners(put(cleanersXPos, nextPos,
                   addTmpMap));
}
```

Monitoring the Size of the Dirty Floor

We are using the active method `run` for monitoring how the size of the floor is decreasing over time.

```
Unit run(){
    while (dirtyFloor != EmptySet){
        floorProgress = Cons(Pair(now(), size(dirtyFloor)), floorProgress);
        await duration(5,5);
    }
}
```

Here **await duration**`(5,5)` is used to suspend the process `run` and only active it for measuring the size of the floor every five times steps .

4.3 The Cleaning Robots

We model the cleaning robots (defined in Section 2.1) as a multi-cleaning robot system. Each cleaning robot follows the CLEAN protocol [31] described in Figure 4.7. In this section we will first explain the user-defined datatypes we use to represent the local environment in the memory of the robots, and later we will explain how a cleaning robot is modelled in the class `CleanerImp` with interface `Cleaner`. We will also explain some of the relevant functions used in the implementation of the CLEAN protocol.

4.3.1 User-defined Datatypes

For representing the local environment, we reuse the datatypes explained in Section 4.2: The local dirty floor is of type `Graph`, the cleaners located in the local neighbourhood are stored in the map of type `CleanersPerPos`, the signals to the intended next position of the cleaners in the local neighbourhood are stored in a map of type `CleanerSignal`, and so forth.

In addition we use datatypes `CleanerPath` and `CLEANPath` for monitoring the cleaners. The list `CleanerPath` records how a cleaner moves around the dirty floor. Here `Pair<Time,Pos>` represents the time when a cleaner moves to a specific position in the dirty floor. The list `CLEANPath` records the time when a specific subtask of the CLEAN algorithm is executed. The triple `Triple<Time,String,Pos>` represents the time, the acronym of the subtask and the current position, respectively.

```
type CleanerPath = List<Pair<Time,Pos>>;
type CLEANPath = List<Triple<Time,String,Pos>>;
```

4.3.2 The Implementation of the Cleaning Robots

The following interface lists all the operations that we use to setup the robots and to retrieve monitoring information.

```
interface Cleaner {
   Unit reset(Pos p0, Environment e, Bool traceClean, Bool tracePath);
   Unit stopped();
}
```

The class `CleanerImp` implements the interface `Cleaner`; the attributes in Table 4.1 are used to represent the internal state of the robots:

`env`	Reference to the environment object
`localDirtyFloor`	Graph containing the local dirty floor
`localCleaners`	Map containing the cleaners in the local neighbourhood
`localSignals`	Map containing the signals to intended next position from cleaners in the local neighbourhood
`initPos`	Initial position of the cleaner
`pos`	Current position of the cleaner
`prevPos`	Previous position of the cleaner
`nextPos`	Next intended position of the cleaner
`resting`	Flag indicating if the cleaner is resting or not
`cleanInLastTour`	Flag indicating if the cleaner had cleaned a position in the last time interval
`waitingSet`	A set of position indicating if the cleaner is waiting for some other cleaners to finish their cycle
`saturatedPerimeter`	A flag indicating if some specific conditions are happening when the cleaners are almost finishing to clean the floor. See subroutine `checkTaskNearCompletion` for more details
`priority`	Priority value.
`lastTimeMove`	Last time the cleaner moved
`timeStep`	Current discrete time
`stop`	Flag indicating if the cleaner has stopped
`cleanerPath`	Monitoring information to see the path for the cleaner over the floor.
`onCleanerPath`	Turn on the cleaner path recording
`pCLEANPath`	Monitoring information to see how the CLEAN protocol had been executed
`onCLEANPath`	Turn on the CLEAN path recording

Table 4.1: Attributes of a cleaning robot

```
class CleanerImp(String name) implements Cleaner{
  Environment env;
  Graph localDirtyFloor = EmptySet;
  CleanersPerPos localCleaners = EmptyMap;
  CleanerSignal localSignals = EmptyMap;
  Pos initPos = Pair(-999,-999);
  Pos pos = Pair(-999,-999);
  Pos prevPos = Pair(-999,-999);
  Pos nextPos = Pair(-999,-999);
  Bool resting = False;
```

```
 Bool cleanInLastTour = False;
 PosSet waitingSet = EmptySet;
 Bool saturatedPerimeter = False;
 Int priority = 0;
 Time lastTimeMove = Time(0);
 Time timeStep = Time(0);
 Bool stop = False;

 CleanerPath cleanerPath = Nil;
 Bool onCleanerPath = False;
 CLEANPath pCLEANPath = Nil;
 Bool onCLEANPath = False;
...
}
```

Reset Method

This method is used to setup the cleaner with initial values for its attributes. It also sends an initial signal to the environment and then starts the CLEAN protocol.

```
Unit reset(Pos p0,Environment e,Bool cLEANPathOn,Bool cleanerPathOn){
     env = e;
     localDirtyFloor = EmptySet;
     localCleaners = EmptyMap;
     localSignals = EmptyMap;
     initPos = p0;
     pos = p0;
     prevPos = left(pos);
     nextPos = Pair(-999,-999);
     resting = False;
     cleanInLastTour = True;
     waitingSet = EmptySet;
     nearCompletion = False;
     saturatedPerimeter = False;
     priority = priority(prevPos,pos);
     lastTimeMove = now();
     timeStep = now();
     stop = False;

     CleanerPath = Nil;
     onCleanerPath = cleanerPathOn;
     pCLEANPath = Nil;
     onCLEANPath = cLEANPathOn

     Fut<Unit> f = env!initSignal(name, pos); f.get;
     this!checkTaskCompleted();
  }
```

Priority. The priority value is calculated using the following function. Here `pos0` is the previous position and `pos1` is the current position.

```
  def Int priority(Pos pos0 ,Pos pos1) =
   case pos1 {
     Pair(x1,y1) => case pos0 {
        Pair(x0,y0) => 2*(x1-x0) + (y1-y0);};};
```

Stop

This method is used to indicate that a cleaner has stopped. It will be called in the main block and it is used as a waiting condition before checking if the goal has been reached. See Section 4.4.

```
Unit stopped(){ await stop; }
```

Sensing the Neighbourhood

This method is used to retrieve information from the environment such as the local dirty floor and the cleaners in the local neighbourhood.

```
Unit sense(){
    Fut<Graph> f1 = env!getLocalDirtyFloor(pos);
    localDirtyFloor = f1.get;
    Fut<CleanersPerPos> f2 = env!getLocalCleaners(pos);
    localCleaners = f2.get;
    Fut<CleanerSignal> f3 =
                env!getLocalSignals(listCleaners(localCleaners));
    localSignals = f3.get;
}
```

Here `pos` is the current position of the cleaner and the function `listCleaners` takes as an argument a map of type `CleanersPerPos` and returns the list of cleaner's names found in that map.

4.3.3 The CLEAN Protocol

The CLEAN protocol [31] is a cyclic algorithm that is divided in a number of subtasks. In our ABS model of the CLEAN protocol, each subtask is implemented as a method which at the end calls the corresponding next subtask creating a sort of cyclic waterfall. Below is a sketch of the protocol in Real-Time ABS. In this sketch we are highlighting the control flow of the different subtasks and the implicit synchronisation.

```
Unit checkTaskCompleted(){
     ... if ...{... stop = True; } else { this!checkTaskNearCompletion(); } }

Unit checkTaskNearCompletion(){
     ... if ...{... stop = True; } ... if (~stop) { this!calcDestination();} }

Unit calcDestination(){ ... this!signalDestination(); }

Unit signalDestination(){...this!calcRestingDependecies(); }

Unit calcRestingDependecies(){ await duration(1/4,1/4); ...
```

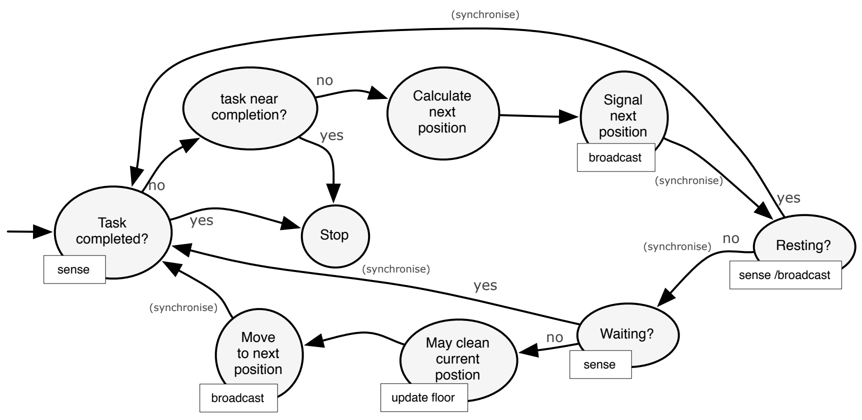


Figure 4.7: Workflow of the CLEAN protocol

```
   if ...{ ... resting = True; await duration(3/4,3/4);
          this!checkTaskCompleted(); } ...
   else { await duration(1/4,1/4); this!calcWaitingSet(); } }

Unit calcWaitingSet(){...
   if (~emptySet(waitingSet)) { await duration(1/2,1/2); this!checkTaskCompleted();}
   else{ this.mayCleanCurrentPos();} }

Unit mayCleanCurrentPos() {
   if ( ~isCritical(pos, localDirtyFloor) && ...) {
        ... env!cleanTile(pos,name); ...}
   ... this.moveToDestination(); }

 Unit moveToDestination(){
     ... env!updatePos ... await duration(1/2,1/2); this!checkTaskCompleted(); }
```

Deviation from the original protocol. The work flow of our proposed model for the CLEAN protocol is depicted in Figure 4.7 while the original workflow is shown in Figure A.1 of Appendix A. We decided to modify the workflow because in the original one there is one state called "is there still enough time?" that we could not directly express with the current semantics of Real-Time ABS. The time semantics of Real-Time ABS states that all the objects do as much as they can before time advances, which means that all the objects should be blocked or suspended before time can advance. In our Real-Time ABS model of the cooperative cleaners, we basically assumed that *there is never enough time* which means that if a cleaning robot decides to rest or to wait then it will be suspended until the discrete time step finishes and will then restart the cycle.

In addition, as we explained in Section 4.1, the CLEAN protocol assumes a sort of implicit barrier synchronisation between the cleaners. We have modelled this implicit barrier synchronisation by introducing in the model **await** statements that suspend the cleaners. These suspensions guarantee that all the cleaners have signalled their desired next position before the resting conditions are checked, that all the cleaners have decided if they are resting or not before the waiting conditions are checked, and that all cleaners execute one coordinated cycle of the CLEAN algorithm per discrete time step.

In the rest of this section we are going to explain the implementation of the different subtasks in the protocol.

Check Task Completed

This method detects cases where all the positions have been cleaned. Here the Boolean function `neighborsDirty` is `True` if at least one of the *8-neighbours* of the current position is dirty.

```
 Unit checkTaskCompleted(){
     timeStep = now();
     if (onCLEANPath) {
          pCLEANPath = Cons(Triple(now(),"CTC",pos),pCLEANPath);
     }
     if (onCleanerPath ) {
          cleanerPath = Cons(Pair(timeStep,pos),cleanerPath);
     }
     this.sense();
     Bool isNeigborhoodDirty = neighborsDirty(eightNbr(pos,localDirtyFloor));
     if (pos == initPos && isNeigborhoodDirty==False )
     {
        if (posIsDirty(pos,localDirtyFloor))
        {
           Fut<Unit> f = env!cleanTile(pos,name,timeStep); f.get;
        }
        stop = True;
        Fut<Unit> f =
           env!updateStatus(name, pos, Stop, timeStep,priority); f.get;
     }
     else {
        this!checkTaskNearCompletion();
     }
  }


def Bool neighborsDirty(Graph g) = ~ emptySet(g);
```

Check Task Almost Completed

This method identifies scenarios in which there are too many cleaning robots blocking each other, preventing the cleaning of the last few positions. Here the Boolean function `checkEightNbsCond` (see Appendix C) is `True` if all the positions in the local dirty floor have at least one cleaner, and the Boolean function `checkSaturatedPerimeter` (see Appendix C) is `True` if every non-critical position in the border of the local dirty floor contains at least two cleaners (see the implementation of the functions `posIsInBorder` and `isCritical` below).

```
 Unit checkTaskNearCompletion(){
     if (onCLEANPath) {
        pCLEANPath = Cons(Triple(now(),"CNC",pos),pCLEANPath)
     ;}
```

```
    if (pos == initPos &&
       checkEightNbsCond(pos,localDirtyFloor,localCleaners)){
       if (posIsDirty(pos,localDirtyFloor))
       {
          Fut<Unit> f = env!cleanTile(pos,name,timeStep); f.get;
       }
       stop = True;
       Fut<Unit> f =
           env!updateStatus(name, pos, Stop, timeStep,priority); f.get;
    }
    saturatedPerimeter = False;
    if (checkSaturatedPerimeter(pos, localDirtyFloor,localCleaners)) {
       saturatedPerimeter = True;
    }
   if (~stop) { this!calcDestination();}
}
```

Deviation from the original protocol. As it can be observed in Appendix A, in the definition of this subtask there is a condition that states: "this is the end of the tour". Since we could not find an explanation in the paper about the meaning of this condition, we decided to investigate further in other publications of the authors. In [6] the authors describe a variation of the cooperative cleaners problem. The variation of the problem involves a deterministic expansion of the dirty floor simulating spreading of contamination or fire, in this paper the authors propose a variation of the CLEAN protocol called SWEEP. The SWEEP protocol also contains a subtask called "Check Near Completion of Mission". We decided to base the implementation of this subtask as it is described in the SWEEP protocol (see Appendix B).

Calculate Next Position

This method signals that the cleaner is active and calculates the next intended position using the function `destination`.

```
Unit calcDestination(){
  if (onCLEANPath) {
     pCLEANPath = Cons(Triple(now(),"CD",pos),pCLEANPath) ;
  }
  resting = False;
  Fut<Unit> f=
      env!updateStatus(name,pos,Active,lastTimeMove,priority); f.get;
  nextPos = destination(prevPos, pos, localDirtyFloor);
  this!signalDestination();
}
```

The Rightmost Position. The function `destination` calculates the next intended position which is the *rightmost neighbour* of the current position. The *rightmost neighbour* of the current position `pos` is defined as follows: starting from the previous position, scan the *4-neighbours* of (x, y) in clockwise order until another boundary position is found.

```
def Pos destination(Pos pp,Pos pos, Graph fullNeighbors) =
   case neighborsDirty(eightNbr(pos,fullNeighbors)) {
     False => pos;
     True => case posIsDirty(tmove(pp, pos),fullNeighbors) &&
                           posIsInBorder(tmove(pp,pos),fullNeighbors){
         True => tmove(pp,pos);
           False => destination(tmove(pp,pos),pos, fullNeighbors);};};

def Pos tmove(Pos pp, Pos p)=
  let (Pos l)=left(p) in
  let (Pos u)=up(p) in
  let (Pos r)=right(p) in
  let (Pos d)=down(p) in
  case pp { l => up(p);
         u => right(p);
         r => down(p);
         d => left(p);};
```

Boundary Position. A position `p` is on the border of the dirty subgraph `g`, if the cardinality of its *8-neighbours* set is less than eight. The function `size` calculates the cardinality of a set.

```
def Bool posIsInBorder(Pos p, Graph g) = size(eightNbr(p,g))<8 ;
```

Deviation from the original protocol. The calculation of the next intended position depends on the previous position `prevPos`. Following the definition of the problem, at time 0 there is no previous position, for that reason, in the original definition of the CLEAN protocol, the first movement of a cleaner is determined by the charts in Figure A.2 of Appendix A. For simplicity we decided instead to artificially mark `prevPos = left(pos)` in the `reset` method and calculate the next intended position as in any other time step.

Signal Next Position

This method signals the intended next position of the cleaner to the environment.

```
 Unit signalDestination(){
   if (onCLEANPath) {
      pCLEANPath = Cons(Triple(now(),"sD",pos),pCLEANPath) ;
   }
     Fut<Unit> f = env!updateDestinationSignal(name, nextPos); f.get;
     this!calcRestingDependecies();
 }
```

Resting Conditions

This method checks if this cleaner should give priority to other cleaners which are located in the same next position and which are signalling to the same position. This part of the algorithm prevents the cleaning robots for grouping into clusters which move together. By applying the "resting policy", the cleaning robot who first entered the position gets to leave the position first and the other cleaning robots rest. If several cleaning robots have entered a position at the same time the robot which is first allowed to leave is determined by the *priority* function. Note that this part of the algorithm is ignored if the perimeter of the local dirty sub-graph is *saturated*. A perimeter is saturated if every non-critical position in the border of the local dirty floor contains at least two cleaners. This special case typically occur when the cleaners are close to reach their goal of cleaning the floor (see the implementation of subtask *check task almost completed*).

The implementation of this method can be summarised as follows: From all cleaners in position pos (except this cleaner), create two groups of cleaners. The first group contains cleaners which signal to the same next position but that moved to position pos before this cleaner, the second group contains cleaners which signal to the same next position, moved at the same time to position pos but have higher priority than this cleaner. If any of these two groups is not empty then the cleaner must rest. If the cleaner is resting then it suspends until the next time interval and restarts the cycle of the algorithm. The different auxiliary functions in the code help to filter the corresponding groups. The implementation of the auxiliary functions can be found in Appendix C.

```
  Unit calcRestingDependecies(){
    await duration(1/4,1/4);
    if (onCLEANPath) {
       pCLEANPath = Cons(Triple(now(),"CRD",pos),pCLEANPath );
    }
    this.sense();
    if (~saturatedPerimeter) {
       Cleaners cleanersMapWithSamePos =
              lookupDefault(localCleaners,pos, EmptyMap);
       cleanersMapWithSamePos =
              removeCleanerName(cleanersMapWithSamePos, name);
       List<String> cleanersWithSamePos =
              keysList(cleanersMapWithSamePos);
       List<String> cleanersWithSamePosSameSignal =
              removingSignalDifPos(localSignals,
                                   cleanersWithSamePos, nextPos);
       List<String> cleanersWithSamePosSameSignalSameTime =
              enterSameTimeAsCleaner(cleanersMapWithSamePos,
                     cleanersWithSamePosSameSignal,lastTimeMove);
       if (enterBeforeCleaner(cleanersMapWithSamePos,
                               cleanersWithSamePosSameSignal,
                               lastTimeMove)||
          isHigherPriority(cleanersMapWithSamePos,
                           cleanersWithSamePosSameSignalSameTime ,
                           priority)){
          resting = True;
          Fut<Unit> f =
             env!updateStatus(name,pos,Resting,lastTimeMove,priority);
          f.get;
          await duration(3/4,3/4);
          this!checkTaskCompleted();
       }
       else {
          await duration(1/4,1/4);
          this!calcWaitingSet();
       }
    }
    else {
       await duration(1/4,1/4);
       this!calcWaitingSet();
   }
}
```

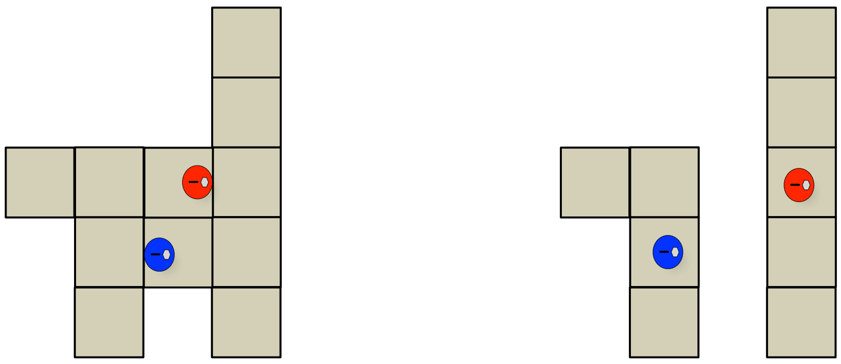


Figure 4.8: Example of a violation of the dirty floor connectivity due to lack of synchronisation when concurrently cleaning two non-critical dirty positions

Waiting Conditions

This method implicitly synchronises this cleaner with the rest of the cleaners in the local neighbourhood. The synchronisation aims to prevent concurrent cleaning of non-critical positions that can damage the connectivity of the dirty subgraph (as depicted in Figure 4.8). In order to avoid concurrent cleaning of adjacent positions this part of the algorithm imposes an order of cleaning between cleaning robots located next to each other. This part of the algorithm ignores cleaners in resting status. The implementation of the method `calcWaitingSet` basically calculates a waiting set `waitingSet` by following the conditions described in Table 5.1 and depicted in Figure 4.9. If `waitingSet` is not empty then this cleaner suspends until the next time interval and restarts the cycle of the algorithm. The different auxiliary functions in the code help to calculate the set `waitingSet`. The implementation of the auxiliary functions can be found in Appendix C.

```
Unit calcWaitingSet(){
  if (onCLEANPath) {
     pCLEANPath = Cons(Triple(now(),"CWD",pos),pCLEANPath );
  }
  waitingSet = EmptySet;
  this.sense();
  Cleaners tmpMap = filterNotMoveThisTime(filterActive(lookupDefault(
                        localCleaners ,left(pos), EmptyMap)),timeStep);
  if (posIsDirty(left(pos),localDirtyFloor) && tmpMap != EmptyMap) {
     waitingSet = insertElement(waitingSet,left(pos));
  }
  tmpMap = filterNotMoveThisTime(filterActive(lookupDefault(
                        localCleaners ,down(pos), EmptyMap)),timeStep);
  if (posIsDirty(down(pos),localDirtyFloor) && tmpMap != EmptyMap) {
      waitingSet = insertElement(waitingSet,down(pos));
  }
  tmpMap = filterNotMoveThisTime(filterActive(lookupDefault(
                        localCleaners ,downleft(pos), EmptyMap)),timeStep);
  if (posIsDirty(downleft(pos),localDirtyFloor) && tmpMap != EmptyMap) {
      waitingSet = insertElement(waitingSet,downleft(pos));
  }
```

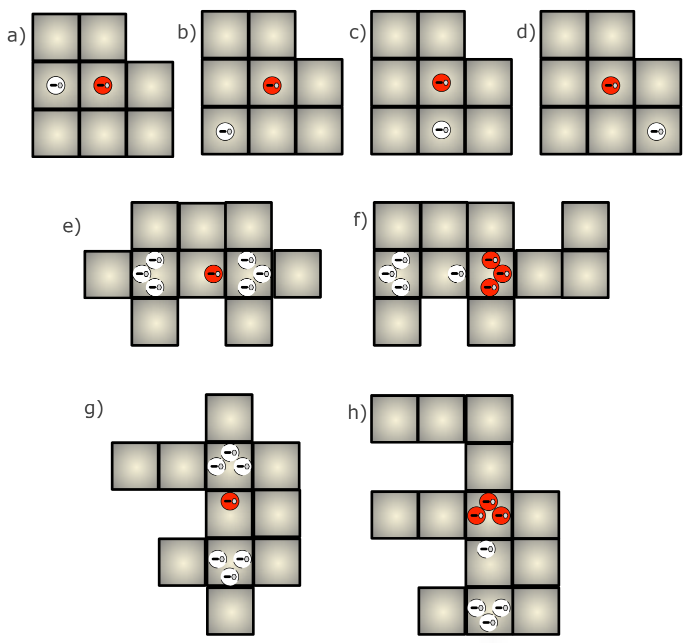


Figure 4.9: Examples of the waiting dependencies. Scenarios a, b, c, d, e, and g add elements to the waiting set. Scenarios f and h remove elements from the waiting set.

```
tmpMap = filterNotMoveThisTime(filterActive(lookupDefault(
                  localCleaners ,downright(pos), EmptyMap)),timeStep);
if (posIsDirty(downright(pos),localDirtyFloor) && tmpMap != EmptyMap) {
   waitingSet = insertElement(waitingSet,downright(pos));
}
if (nextPos == right(pos) ) {
   Cleaners tmpMapRight =
      filterActive(lookupDefault(localCleaners,right(pos),EmptyMap));
   Cleaners tmpMapLeft =
      filterActive(lookupDefault(localCleaners,left(pos),EmptyMap));
   Cleaners tmpMapUp =
      filterActive(lookupDefault(localCleaners,up(pos),EmptyMap));
   if (thereIsCleanerNotSignalingTo(tmpMapRight,localSignals, pos)) {
     if (thereIsNoCleanerWithSignalTo(tmpMapLeft,localSignals, pos)
         && tmpMapUp==EmptyMap &&
         thereIsNoCleanerWithSignalTo(tmpMapRight,localSignals, pos)){
            waitingSet = insertElement(waitingSet,right(pos));
     }
   }
 }
 if (nextPos != left(pos) ) {
   tmpMap =
      filterActive(lookupDefault(localCleaners,pos,
             EmptyMap));
   Cleaners tmpMapLeft =
      filterActive(lookupDefault(localCleaners,left(pos),
             EmptyMap));
```

		$W = \emptyset$
a)	**If**	$(x-1,y) \in$ *boundary* of $F_t \wedge (x-1,y)$ contains other active cleaning robots which did not move in the current time interval.
	Then	$W = W \cup \{(x-1,y)\}$
b)	**If**	$(x-1,y-1) \in$ *boundary* of $F_t \wedge (x-1,y-1)$ contains other active cleaning robots which did not move in the current time interval.
	Then	$W = W \cup \{(x-1,y-1)\}$
c)	**If**	$(x,y-1) \in$ *boundary* of $F_t \wedge (x,y-1)$ contains other active cleaning robots which did not move in the current time interval.
	Then	$W = W \cup \{(x,y-1)\}$
d)	**If**	$(x+1,y-1) \in$ *boundary* of $F_t \wedge (x+1,y-1)$ contains other active cleaning robots which did not move in the current time interval.
	Then	$W = W \cup \{(x+1,y-1)\}$
e)	**If**	next position is $(x+1,y) \wedge (x+1,y)$ contains an active cleaning robot $j \wedge$ next position of j is not $(x,y) \wedge (x-1,y)$ does not contain an active cleaning robot l where next position of l is $(x,y) \wedge (x,y+1)$ does not contain any active cleaning robots $\wedge (x+1,y)$ does not contain an active cleaning robot n where next position of n is (x,y)
	Then	$W = W \cup \{(x+1,y)\}$
f)	**If**	next position is not $(x-1,y) \wedge (x-1,y)$ contains an active cleaning robot $j \wedge$ next position of j is $(x,y) \wedge (x-2,y)$ does not contain an active cleaning robot l where next position of l is $(x-1,y) \wedge (x-1,y-1)$ does not contain any active cleaning robots $\wedge (x,y)$ does not contain an active cleaning robot n where next position of n is $(x-1,y)$
	Then	$W = W - \{(x-1,y)\}$
g)	**If**	next position is $(x,y+1) \wedge (x,y+1)$ contains an active cleaning robot $j \wedge$ next position of j is not $(x,y) \wedge (x,y-1)$ does not contain an active cleaning robot l where next position of l is $(x,y) \wedge (x+1,y)$ does not contain any active cleaning robots $\wedge (x,y+1)$ does not contain an active cleaning robot n where next position of n is (x,y)
	Then	$W = W \cup \{(x,y+1)\}$
h)	**If**	next position is not $(x,y-1) \wedge (x,y-1)$ contains an active cleaning robot $j \wedge$ next position of j is $(x,y) \wedge (x,y-2)$ does not contain an active cleaning robot l where next position of l is $(x,y-1) \wedge (x+1,y-1)$ does not contain any active cleaning robots $\wedge (x,y)$ does not contain an active cleaning robot n where next position of n is $(x,y-1)$
	Then	$W = W - \{(x,y-1)\}$
	If	$W \neq \emptyset$ **Then** wait

Table 4.2: Our adaptation for the computation of waiting dependencies for position (x,y). Here W is the waiting set

```
    Cleaners tmpMap2Left =
       filterActive(lookupDefault(localCleaners,left(left(pos)),
              EmptyMap));
    Cleaners tmpMapUpLeft =
       filterActive(lookupDefault(localCleaners,upleft(pos),
              EmptyMap));
    if (thereIsCleanerSignalingTo(tmpMapLeft,localSignals, pos)) {
      if (thereIsNoCleanerWithSignalTo(tmpMap2Left,localSignals,
             left(pos)) && tmpMapUpLeft==EmptyMap &&
         thereIsNoCleanerWithSignalTo(tmpMap,localSignals,left(pos))){
             waitingSet = remove(waitingSet,left(pos));
         }
      }
  }
  if (nextPos == up(pos) ) {
    Cleaners tmpMapUp =
        filterActive(lookupDefault(localCleaners,up(pos),EmptyMap));
    Cleaners tmpMapDown =
        filterActive(lookupDefault(localCleaners,down(pos),EmptyMap));
    Cleaners tmpMapRight =
        filterActive(lookupDefault(localCleaners,right(pos),EmptyMap));
    if (thereIsCleanerNotSignalingTo(tmpMapUp,localSignals, pos)) {
      if (thereIsNoCleanerWithSignalTo(tmpMapDown,localSignals, pos)
         && tmpMapRight==EmptyMap &&
         thereIsNoCleanerWithSignalTo(tmpMapUp,localSignals, pos)) {
            waitingSet = insertElement(waitingSet,up(pos));
        }
      }
 }
 if (nextPos != down(pos) ) {
    tmpMap =
       filterActive(lookupDefault(localCleaners,pos,
             EmptyMap));
    Cleaners tmpMapDown =
       filterActive(lookupDefault(localCleaners,down(pos),
             EmptyMap));
    Cleaners tmpMap2Down =
       filterActive(lookupDefault(localCleaners,down(down(pos)),
             EmptyMap));
    Cleaners tmpMapDownRight =
       filterActive(lookupDefault(localCleaners,downright(pos),
             EmptyMap));
    if (thereIsCleanerSignalingTo(tmpMapDown,localSignals, pos)) {
      if (thereIsNoCleanerWithSignalTo(tmpMap2Down,localSignals,
         down(pos)) && tmpMapDownRight==EmptyMap &&
         thereIsNoCleanerWithSignalTo(tmpMap,localSignals, down(pos))){
           waitingSet = remove(waitingSet,down(pos));
        }
      }
 }
 if (~emptySet(waitingSet)) {
     await duration(1/2,1/2);
     this!checkTaskCompleted();
 }
  else{
     this.mayCleanCurrentPos();
 }
```

Deviation from the original protocol. As it can be observed in Appendix A (steps 6.f and 6.g of the CLEAN protocol), the original algorithm allows a cleaning robot to read and write on the internal state of other cleaning robots while calculating its waiting set, for this reason we decided to adapt this part of the algorithm in order to keep the autonomy of the cleaning robots. This adaptation intends to preserve the original final result of the waiting sets. Table 5.1 summarises the new implementation of this part of the algorithm.

May Clean Current Position

This method helps this cleaner to decide if its current position should be cleaned. The current position `pos` is cleaned if it is not the initial position, it is not critical, the next intended position is dirty and there are no other cleaners located in `pos`.

```
Unit mayCleanCurrentPos() {
   if (onCLEANPath) {
      pCLEANPath = Cons(Triple(now(),"MCCP",pos),pCLEANPath );
   }
   if (pos != initPos) {
      if (~isCritical(pos, localDirtyFloor)
         && posIsDirty(nextPos,localDirtyFloor)
         && removeKey(lookupDefault(localCleaners,pos,EmptyMap),name)
                        ==EmptyMap) {
         Fut<Unit> f = env!cleanTile(pos,name,timeStep); f.get;
         cleanInLastTour = True;
      }
      else {
         cleanInLastTour = False;
      }
   }
   if (posIsDirty(nextPos,localDirtyFloor) || nextPos == initPos){
      this.moveToDestination();
   }
   else{
      await duration(1/2,1/2);
      this!checkTaskCompleted();
   }
}
```

Vertex Criticality. As defined in Section 2.1, if a position `p` is removed from the dirty subgraph `g`, such that `g` is no longer a *single connected component* then `p` is a critical position.

The function `isCritical` checks if a position is critical, by checking if the *8-neighbours* of `p` do not form a *single connected component*. Figure 4.10 shows examples of a critical and a non-critical position, respectively.

```
def Bool isCritical(Pos p, Graph g)=
        ~isSingleConnectedComponnet(eightNbr(p,g));
```

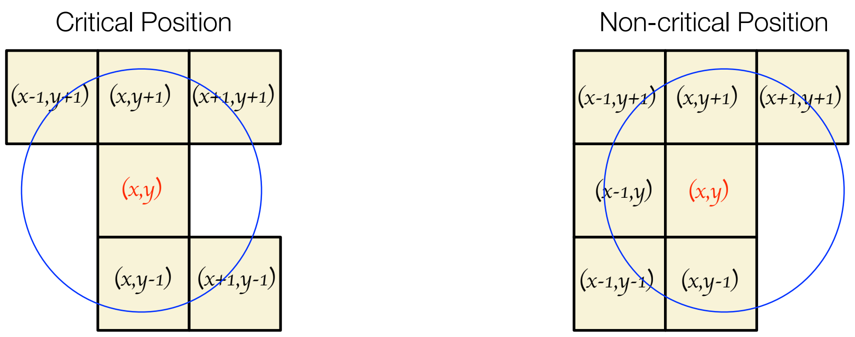


Figure 4.10: Examples for critical (to the left) and non-critical (to the right) positions.

Move to Next Position

This method implements how the cleaner moves to the intended next position.

```
Unit moveToDestination(){
    if (onCLEANPath) {
       pCLEANPath = Cons(Triple(now(),"MD",pos),pCLEANPath);
    }
    lastTimeMove = timeStep;
    priority = priority(pos,nextPos);
    Fut<Unit> f =
       env!updatePos(name,pos,nextPos, Active,lastTimeMove,priority);
    f.get;
    f = env!stopSignalDestination(name); f.get;
    prevPos = pos;
    pos = nextPos;
    await duration(1/2,1/2);
    this!checkTaskCompleted();
 }
```

4.4 The Initial Configuration

In this section we explain how we use the main block to build different shapes of floors as well as to start the algorithm with different numbers of cleaners. A simple initial configuration is depicted in Figure 4.11. In this simple setup the floor is a five per five square, there is only one cleaner and the initial position is $(1, 1)$. We implement this configuration in the main block as follows:

```
 {
     // Create floor and environment
     Graph initialDirtyFloor = makeFloor(1,5,1,5,EmptySet);
     assert isSingleConnectedComponnet(initialDirtyFloor);
     Environment env = new cog EnvironmentImp(initialDirtyFloor);

     // Choose initial position
     Pos p0 = Pair(1,1);
```

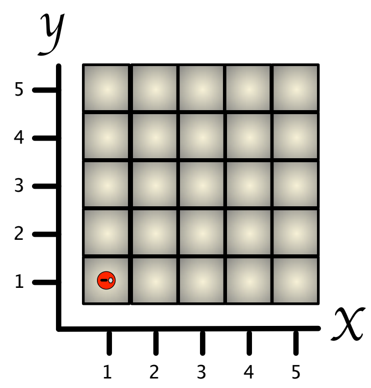


Figure 4.11: A simple initial configuration. Here the floor is a square and there is only one cleaner

```
    // Create cleaner
    Cleaner c1 = new cog CleanerImp("RED");

    // Start protocol
    c1!reset(p0, env,True,True);
    Fut<Unit> finishC1 = c1!stopped();
    await finishC1? ;

    // Check if goal has been successfully reached
    Graph finalDirtyFloor = await env!getDirtyFloor();
    Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor);
}
```

Observe that after the cleaner stops, we are checking the status of the floor using the function `isFloorDirty`. This function is `False` if the final dirty floor is an empty set.

```
  def Bool isFloorDirty(Graph graph) = ~emptySet(graph) ;
```

Building a Floor. The function `makeFloor` builds a floor given a range of values for x and range of values for y. Figure 4.11 shows a floor where the range of x is $[1,5]$ and the range of y is also $[1,5]$. The function `makeFloor` uses the function `insertRow` which builds rows of positions.

```
def Graph makeFloor(Int x, Int limX, Int y, Int limY, Graph floor)=
  case y {
    limY => insertRow(x,limX,y,floor);
    _ => makeFloor(x,limX,y+1,limY,insertRow(x,limX,y,floor));};

def Graph insertRow(Int x, Int limX, Int y, Graph floor)=
  case x {
    limX => Insert(Pair(x,y),floor);
    _ => insertRow(x+1,limX,y,Insert(Pair(x,y),floor));};
```

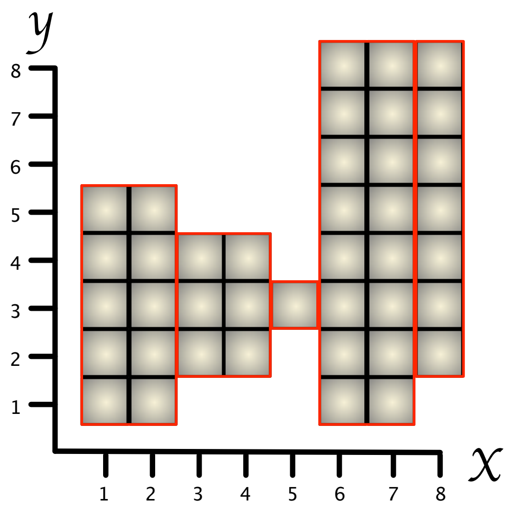


Figure 4.12: Examples of how to generate a floor by composing rectangles.

More complex floor shapes can be constructed by composing rectangles as it is depicted in Figure 4.12. The following code builds the floor shown in Figure 4.12. It recursively calls the function `makeFloor`.

```
Graph graphFloor = makeFloor(1,2,1,5,
                   makeFloor(3,4,2,4,
                   makeFloor(5,5,3,3,
                   makeFloor(6,7,1,8,
                   makeFloor(8,8,2,8, EmptySet)))));
```

Below is the implementation of an initial configuration with the floor depicted in Figure 4.12, with two cleaners and initial position $(2,5)$. Observe that in order to avoid that cleaners follow the same path, there is an interval of two time units between each cleaner starting the CLEAN algorithm. For other initial configurations see Appendix C.

```
{
   // Create floor and environment
   Graph initialDirtyFloor = makeFloor(1,2,1,5,
                             makeFloor(3,4,2,4,
                             makeFloor(5,5,3,3,
                             makeFloor(6,7,1,8,
                             makeFloor(8,8,2,8, EmptySet)))));
    assert isSingleConnectedComponnet(initialDirtyFloor);
    Environment env = new cog EnvironmentImp(initialDirtyFloor);

    // Choose initial position
    Pos p0 = Pair(2,5);

    // Create cleaners
    Cleaner c1 = new cog CleanerImp("RED");
    Cleaner c2 = new cog CleanerImp("BLUE");
    // Start protocol
    c1!reset(p0, env,True,True);
    Fut<Unit> finishC1 = c1!stopped();
    await duration(2,2);
    c2!reset(p0, env,True,True);
    Fut<Unit> finishC2 = c2!stopped();
    await finishC1? ;
    await finishC2? ;

    // Check if goal has been successfully reached
    Graph finalDirtyFloor = await env!getDirtyFloor();
    Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor);
}
```

Chapter 5

Analysis and Simulations

In order to investigate the timing behaviour of specific scenarios in our model of the cooperative cleaners, we are using the interpreter for Real-Time ABS models in Maude [15] where we are running simulations.

As explained in Chapter 4, our current model has been improved after a series of iterations using this tool. Simulations have helped us to understand and improve the model, and have helped us to catch errors related to the correct computation of the different functions, the correct manipulation of the data in the model, a number of safety properties that we check by introducing **assert** statements in the model and finally to partially check the correct behaviour of the CLEAN protocol.

There are still misbehaviours in the model that we manage to identify and understand. However, due to the deviations from the original CLEAN protocol that are explained in Chapter 4, we must be careful to relate these misbehaviours in our model to the original protocol. It remains as a future work to figure out a suitable solution for them.

In this chapter we are going to concentrate on analysing the behaviour of our model of the CLEAN protocol, and for that we are presenting simulation results for explaining what is behaving as expected in the model, what is not behaving as expected in the model, and we will explain when and why our model is not behaving as expected. For a better understanding of the simulations, we are going to depict scenarios with a simple five per five square floor with one, two and three cleaners. We are also going to summarise results for other scenarios.

For the analysis along this chapter we are using the monitoring information recorded in the cleaners and in the environment while running simulations. We manually analysed and organised this information in order to get a sort of visual representation out of it.

5.1 Analysing the Behaviour of the CLEAN Protocol Using Simulations

In this section we are going to explain the behaviour of our implementation of the CLEAN protocol.

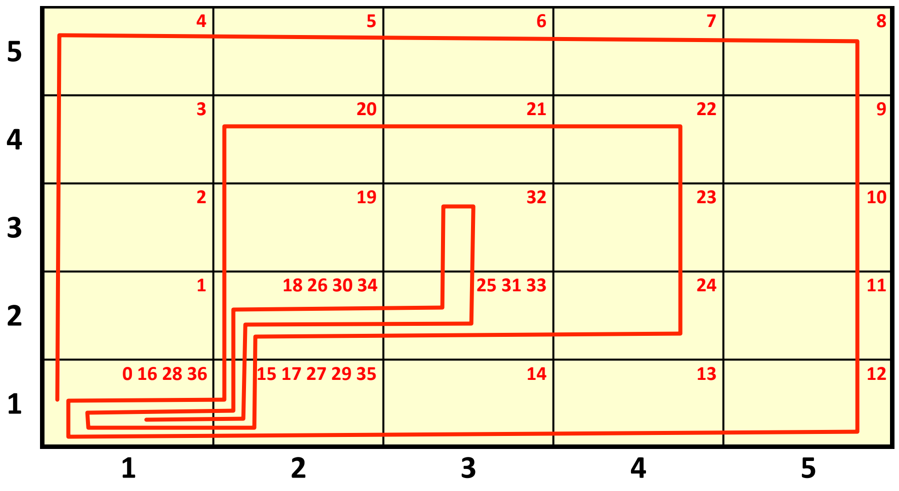
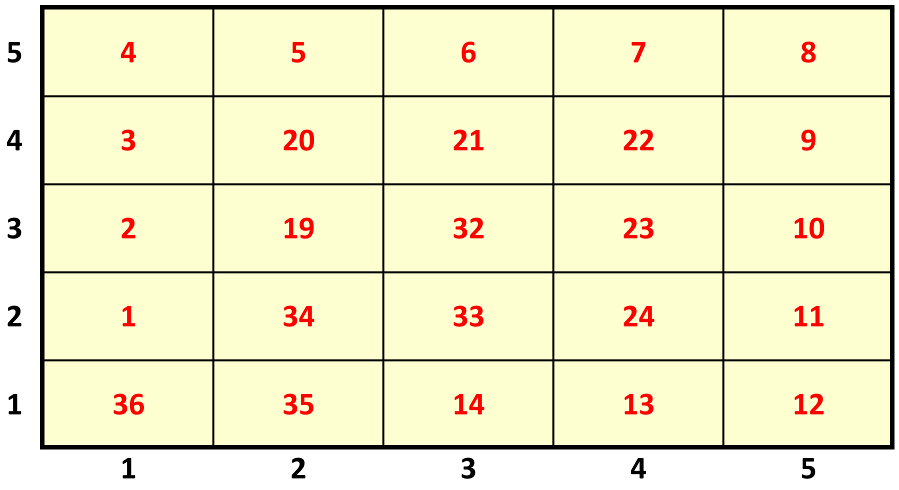


Figure 5.1: Path for only one cleaner on a five per five square floor



5	4	5	6	7	8
4	3	20	21	22	9
3	2	19	32	23	10
2	1	34	33	24	11
1	36	35	14	13	12
	1	2	3	4	5

Figure 5.2: Progress cleaning of a five per five square floor using one cleaner

5.1.1 Five per Five Square Floor with One Cleaner

In this scenario we are depicting a simple simulation where one cleaner, called RED, follows the CLEAN protocol in a five per five square floor, here we let the initial position be $(1, 1)$.

Figure 5.1 depicts how the cleaner RED moved on the floor, in this figure the red line represents the path and the numbers located at the top-right of each position represent the time steps in which cleaner RED was in that position. Figure 5.2 depicts how the five per five square floor was cleaned over time, in this figure the numbers in each position represent the time when a position was cleaned.

Observing both figures we can for example interpret that cleaner RED was in position $(2, 2)$ at time steps 18, 26, 30 and 34 and that it cleaned position $(2, 2)$ at time step 34. Furthermore, we can observe that the cleaner RED behaved as expected, which means that at each time step, cleaner RED cleaned its current position if it was not critical, and moved to the next position. The movement created the effect of a clockwise traversal along the boundary of the dirty floor. As a result, cleaner RED peeled layers from the boundary of the dirty floor, until it was cleaned entirely. In addition cleaner RED started and finished in position $(1, 1)$, cleaner RED only moved along the border of the remaining dirty floor, the connectivity of the remaining dirty floor was preserved over time and that cleaner RED only stopped upon completing its mission.

5.1.2 Five per Five Square Floor with Two Cleaners

In this scenario we are depicting a simulation where two cleaners, called RED and BLUE, follow the CLEAN protocol in a five per five square floor, here as in the previous scenario we let the initial position be $(1, 1)$.

Figure 5.3 and Figure 5.4 depict how cleaners RED and BLUE moved on the floor, respectively. In these figures the red line represents the path of cleaner RED and the blue line represents the path of cleaner BLUE. The numbers located at the top-right of each position represent the time steps in which cleaners RED and BLUE were in that position, respectively. Figure 5.5 depicts how the five per five square floor was cleaned over time, in this figure the numbers in each position represent the time when a position was cleaned, the red numbers are the positions cleaned by cleaner RED, and the blue numbers are the positions cleaned by cleaner BLUE.

Observing these three figures we can interpret the behaviour of the cleaners. As expected, the cleaners moved creating the effect of a clockwise traversal along the boundary of the dirty floor, the cleaners peeled layers from the boundary of the dirty floor until it was cleaned entirely, the cleaners started and finished in position $(1, 1)$, the cleaners only moved along the border of the remaining dirty floor, the connectivity of the remaining dirty floor was preserved over time and the cleaners only stopped upon completing their mission. In addition, we can observe from Figure 5.5 that there were time steps with concurrent cleaning (see positions with a circle around the time step). But it was also expected that each time step each cleaner cleaned its current position if it was not critical and as we can observe from Figure 5.3 and from Figure 5.5 that was not the case (for example, see positions $(3, 5)$ and $(4, 5)$, these positions could have been cleaned by cleaner RED before time 9 and 11, respectively). We can also observe that between the time step intervals $[6, 13]$ and $[18, 30]$ cleaner RED did not clean any position. This misbehaviour mainly affects the poor performance of the cleaning task, performance can be improved if there is more concurrent cleaning of positions.

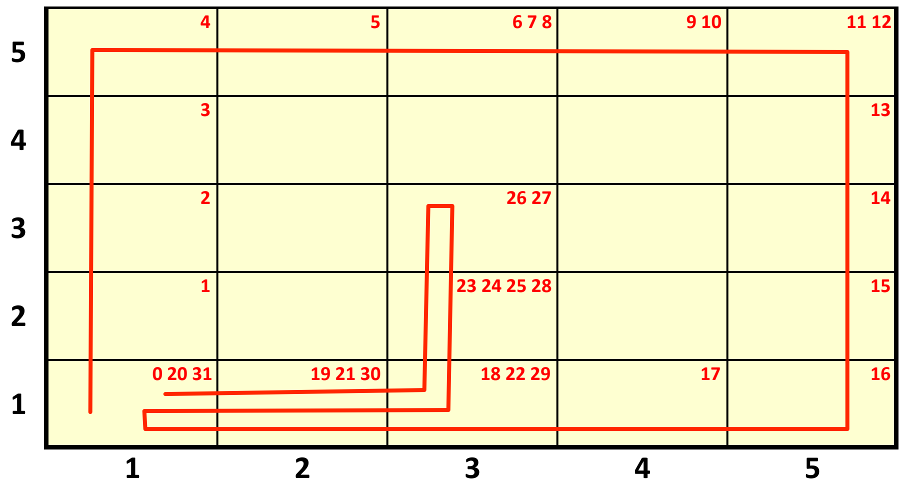


Figure 5.3: Path for cleaner RED on a five per five square floor with two cleaners

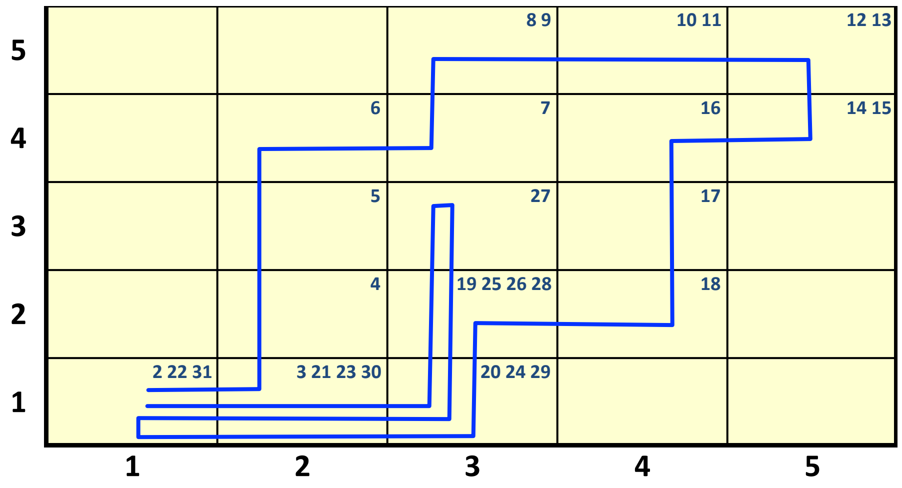


Figure 5.4: Path for cleaner BLUE on a five per five square floor with two cleaners

5.1.3 Five per Five Square Floor with Three Cleaners

In this scenario we are depicting a simulation where three cleaners, called RED, BLUE and GREEN, follow the CLEAN protocol in a five per five square floor, here as in the previous scenarios we let the initial position be $(1, 1)$.

Figure 5.6, Figure 5.7 and Figure 5.8 depict how the cleaners RED, BLUE and GREEN were moving on the floor, respectively. In these figures the red line represents the path of cleaner RED, the blue line represents the path of cleaner BLUE and the green line represents the path of cleaner GREEN. The numbers located at the top-right of each position represent the time steps in which cleaners RED, BLUE and GREEN were in that position, respectively. Figure 5.9 depicts how the five per five square floor was partially cleaned over time, in this

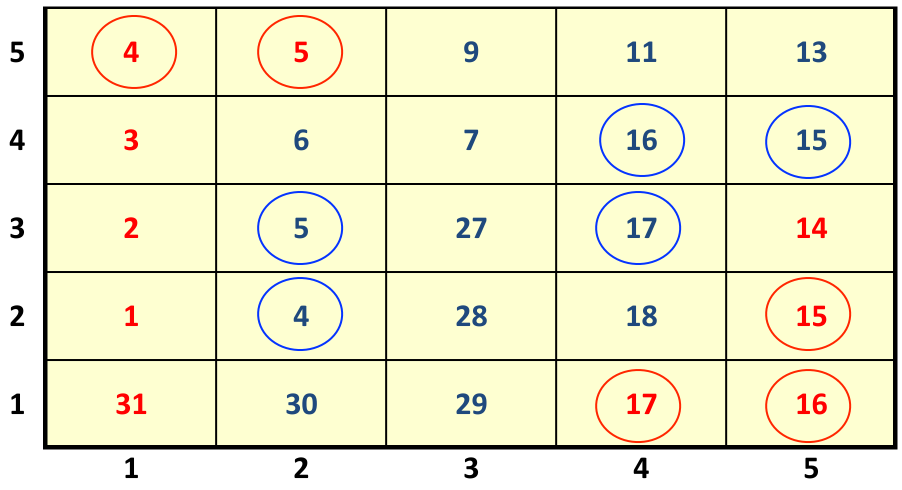


Figure 5.5: Progress cleaning of a five per five square floor using two cleaners, where circles indicate concurrent cleaning.

figure the numbers in each position represent the time when a position was cleaned and the dark positions without a number represent dirty positions. The red numbers are the positions cleaned by cleaner RED, the blue numbers are the positions cleaned by cleaner BLUE and the green numbers are the positions cleaned by cleaner GREEN.

Observing these four figures we can interpret the behaviour of the cleaners. As expected, the cleaners move creating the effect of a clockwise traversal along the boundary of the dirty floor, the cleaners peeled layers from the boundary of the dirty floor, they only moved along the border of the remaining dirty floor and the connectivity of the remaining dirty floor was preserved over time. In addition, we can observe from Figure 5.9 that there were time steps with concurrent cleaning (see positions with a circle around the time step), but only two cleaners were cleaning concurrently at the same time. Observe that these four figures depict simulations until time step 32 only.

In this scenario we detect a livelock, which means that from time step 24 the cleaners move around the remaining dirty graph without managing to clean any positions and as a consequence they never stop. We will explain in detail this livelock detection in Section 5.2.

Similar to the simulation with two cleaners, in this simulation it was not the case that at each time step each cleaner cleaned its current position if it was non-critical, which contributed to the poor perforce along the cleaning task.

5.2 Livelock Detection

In the previous section, we describes simulations in a five per five square floor with one, two and three cleaners. While running the simulation with three cleaners, we detected a livelock which did not let the three cleaners reach their goal but instead enter into an infinite cycle. In this sections we explain this infinite cyclic misbehaviour.

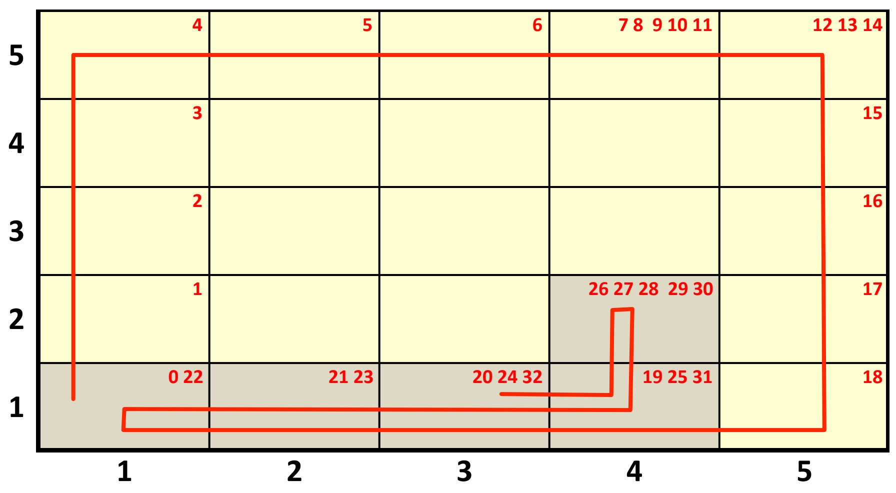


Figure 5.6: Path for cleaner RED on a five per five square floor with three cleaners. The path is depicted until time step 32. Here the darker positions depict dirty positions.

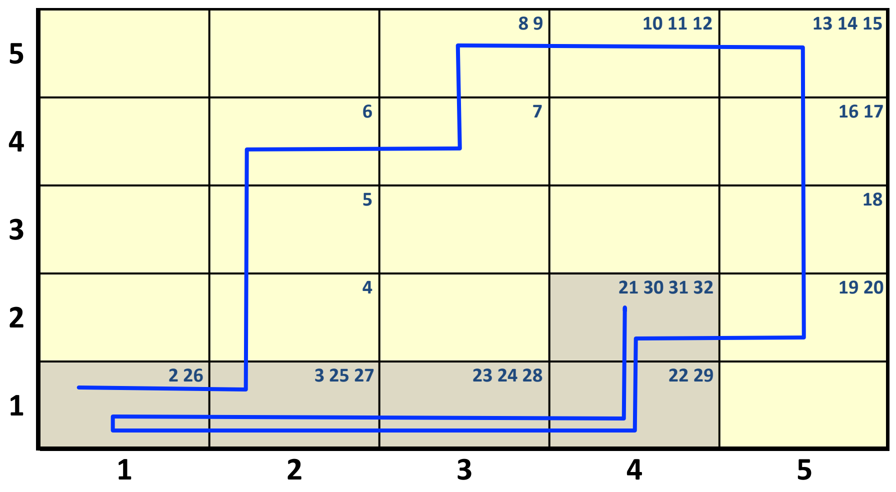


Figure 5.7: Path for cleaner BLUE on a five per five square floor with three cleaners. The path is depicted until time step 32. Here the darker positions depict dirty positions.

Table 5.1 summarises the positions of cleaners RED, BLUE and GREEN between time interval 24–38 in the simulation. In this table we can observe that the three cleaners show the same cyclic pattern interchangeably. Observe that at some point a cleaner remains in position $(4, 2)$ for five time steps, then it moves gradually from $(4, 1)$ to $(1, 1)$ and after that it goes gradually back to $(4, 1)$ to end up again in $(4, 2)$ where it starts a new cycle.

Moreover in Figure 5.10 we graphically depict the positions of the three cleaners on the floor between time steps 24–28 in the simulation. At time step 24, cleaner GREEN is at position $(4, 2)$ while cleaners RED and BLUE are at position $(3, 1)$. According to the conditions on Table 5.1 and depicted in Figure 4.9, cleaner GREEN is waiting for cleaners RED and

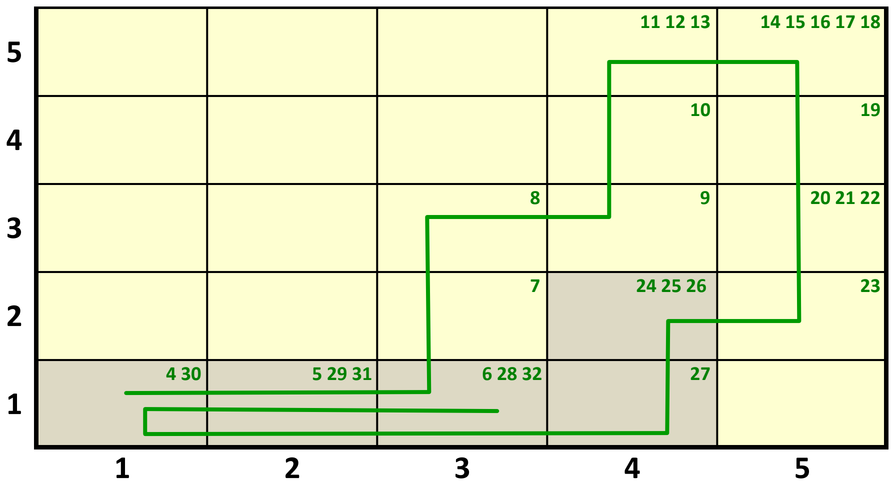


Figure 5.8: Path for cleaner GREEN on a five per five square floor with three cleaners. The path is depicted until time step 32. Here the darker positions depict dirty positions.

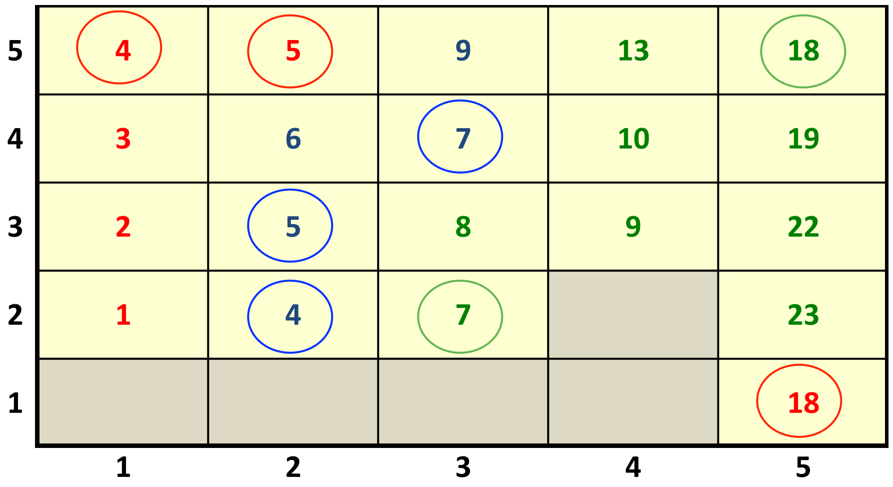


Figure 5.9: Progress cleaning of a five per five square floor using three cleaners. The progress is depicted until time step 32. Here the darker positions depict dirty positions.

BLUE and as a consequence cleaner GREEN restarts its CLEAN protocol cycle and does not clean and move. The position $(3,1)$ is critical so it is not cleaned. At time step 25, cleaner GREEN remains at position $(4,2)$, cleaner BLUE is at position $(2,1)$ and cleaner RED is at position $(4,1)$. In this case cleaner GREEN is again waiting for cleaner RED and as a consequence cleaner GREEN restarts its CLEAN protocol cycle and does not clean and move. The positions $(2,1)$ and $(4,1)$ are critical so they are not cleaned. At time step 26, cleaners RED and GREEN are at position $(4,2)$ and cleaner BLUE is at position $(1,1)$. Since position $(1,1)$ is the initial position and should be the last position to be cleaned, it can not cleaned in this time step. There are two cleaners in the position $(4,2)$. Following the resting conditions, cleaner GREEN moves while cleaner RED rests and restarts its CLEAN protocol

Time	RED	BLUE	GREEN
24	(3,1)	(3,1)	(4,2)
25	(4,1)	(2,1)	(4,2)
26	(4,2)	(1,1)	(4,2)
27	(4,2)	(2,1)	(4,1)
28	(4,2)	(3,1)	(3,1)
29	(4,2)	(4,1)	(2,1)
30	(4,2)	(4,2)	(1,1)
31	(4,1)	(4,2)	(2,1)
32	(3,1)	(4,2)	(3,1)
33	(2,1)	(4,2)	(4,1)
34	(1,1)	(4,2)	(4,2)
35	(2,1)	(4,1)	(4,2)
36	(3,1)	(3,1)	(4,2)
37	(4,1)	(2,1)	(4,2)
38	(4,2)	(1,1)	(4,2)

Table 5.1: Summary of positions for a floor with three cleaners between time 24–38

cycle. As a consequence, position $(4,2)$ is not cleaned. At time step 27, cleaner RED remains in position $(4,2)$, cleaner GREEN is at position $(4,1)$ and cleaner BLUE is at position $(2,1)$. In this time step cleaner RED is waiting for cleaner GREEN and as a consequence cleaner RED restarts its CLEAN protocol cycle and does not clean and move. The positions $(2,1)$ and $(4,1)$ are critical so they are not cleaned. Finally at time 28 we are exactly in the same scenario as at time 24 (except for a permutation between the cleaners). We can conclude from Table 5.1 and from Figure 5.10 that the position $(4,2)$ could not be cleaned mainly due to the waiting conditions. In Section 5.4 we come back to this topic and give some suggestions for how this misbehaviour could be overcome.

5.3 Summary of Simulation Results

In this section we summarise and compare different simulation results. Table 5.2 summarises simulations for a five per five square floor with one, two and three cleaners, for a ten per ten square floor with one, two and three cleaners, and for a twenty per twenty square floor with one and two cleaners. In these simulations we are mainly interested in the the total number of time steps it took to clean a floor with the different scenarions. We can observe that for the five per five square and for the ten per ten square floor (ignoring the livelock when using three cleaners) there is not much improvement in time when increasing the number of cleaners and for the twenty per twenty square floor it took *more time* to clean the floor with two cleaners than with one cleaner. We believe that the reason for this poor performance is that while running the CLEAN protocol, there are too many cases where waiting dependencies apply and as a consequence many delays are unnecessarily introduced. Figure 5.11 depicts the five per five square floor with two cleaners at time step 7, in this figure the red line represents the path of cleaner RED and the blue line represents

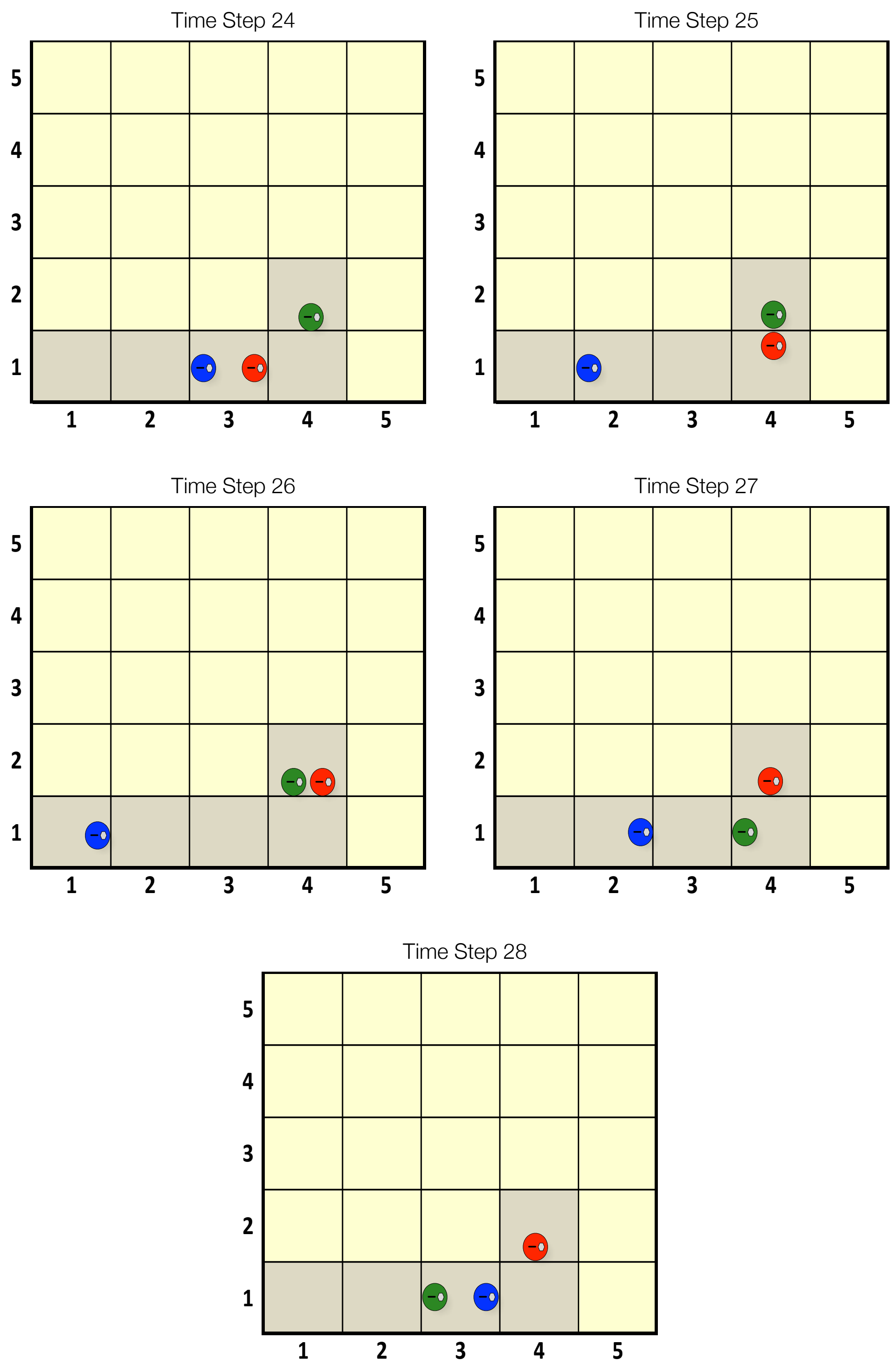


Figure 5.10: Positions of cleaners RED, BLUE and GREEN in a five per five square floor between time steps 24–28.

5x5 (size = 25)	**Time**
1 cleaner	36
2 cleaners	31
3 cleaners	(before livelock) 23
10x10 (size = 100)	
1 cleaner	140
2 cleaners	129
3 cleaners	(before livelock) 83
20x20 (size = 400)	
1 cleaner	580
2 cleaners	587

Table 5.2: Summary of simulation results.

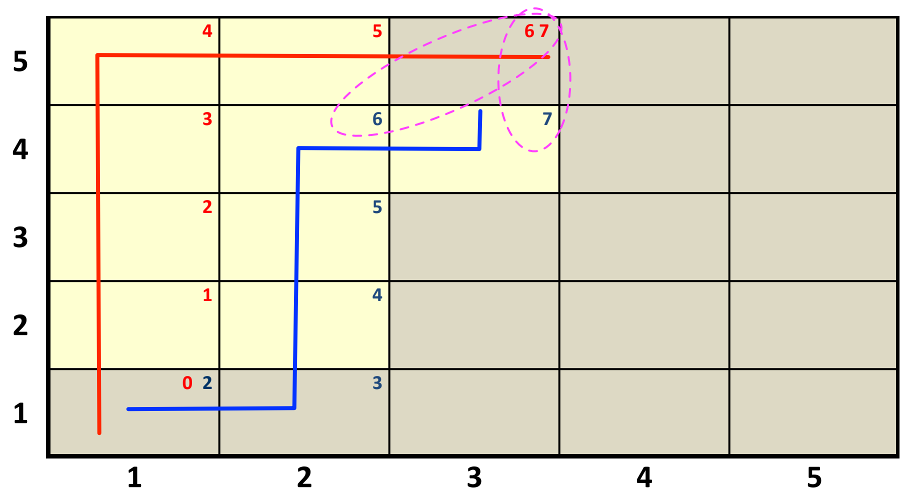


Figure 5.11: Progress cleaning of a five per five square floor using two cleaners. The progress is depicted until time step 7. Here the darker positions depict dirty positions.

the path of cleaner BLUE. The numbers located at the top-right of each position represent the time steps in which cleaners RED and BLUE were in that position, respectively. The dark positions represent dirty positions. As we can observe from this figure, position $(3, 5)$ was not cleaned at time step 6 and will not be cleaned at time step 7 because cleaner RED waits for cleaner BLUE (see Table 5.1 and Figure 4.9), we can also see from Figure 5.5 that position $(3, 5)$ is finally cleaned by cleaner BLUE at time step 9, when it could have been cleaned by cleaner RED at time step 6 if unnecessary delays would not have been introduced. In Section 5.4 we return to this topic and give some suggestions for how we could remove these unnecessary delays.

5x5 (size = 25)	Time
1 cleaner	36
2 cleaners	26
3 cleaners	18

Table 5.3: Simulation results without waiting conditions subtask.

5.4 Discussion of Possible Further Refinements in the Model

In this section we discuss possibilities for how our model could be further refined. We would like to conduct our discussion by answering the following three questions:

1. How could the livelock be eliminated from the model?
2. How could the unnecessary delays be removed from the model?
3. Is the design of the model prepared to support fault-tolerance?

In relation to the first two questions, as described in Sections 5.2 and 5.3, the main problem that the analysis of the simulations helped to detect in our current model is related to the waiting conditions subtask. As an experiment to see if there is a possibility to refine our model by changing the waiting conditions model, we decided to first run some simulations where we did not calculate the waiting set but instead assume that the waiting set is always empty and observe how the cleaners behave.

Figure 5.12 and Figure 5.13 depict how the five per five square floor were cleaned over time with two and three cleaners, respectively. In these two simulations we ignore the method that calculates the waiting conditions by jumping from calculating the *resting conditions* subtask to the *may clean current position* subtask. As before the numbers in each position of the figures represent the time when a position was cleaned, the red numbers are the positions cleaned by cleaner RED, the blue numbers are the positions cleaned by cleaner BLUE, and the green numbers are the positions cleaned by cleaner GREEN. As we can observe in these two figures, there are many positions that were cleaned twice. A characteristic of these positions is that the time steps are consecutive. We believe that this misbehaviour is triggered by race conditions introduced as a side effect of removing the *waiting condition* subtask.

One positive outcome from this experiment is the improvement in the performance of the cleaners. Table 5.3 shows a summary of simulations in a five per five square floor with one, two and three cleaners, ignoring waiting conditions. As we can observe there is a remarkable improvement in the total time required to complete the task. A second positive outcome is that we did not detect the livelock when we ran the simulation with three cleaners.

After finishing this experiment our next question is related to our implementation of the *waiting condition* subtask and what we would need to figure out in order to have a faithful implementation. To answer these questions let us first go back to the definition of the problem in Chapter 2 and to the original CLEAN protocol in Appendix A. In the definition

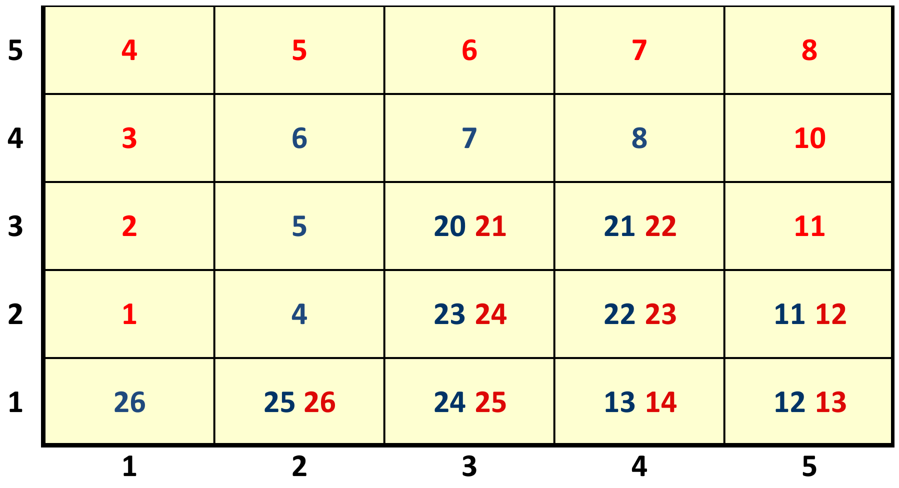


Figure 5.12: Progress cleaning of a five per five square floor using two cleaners and ignoring waiting conditions subtask.

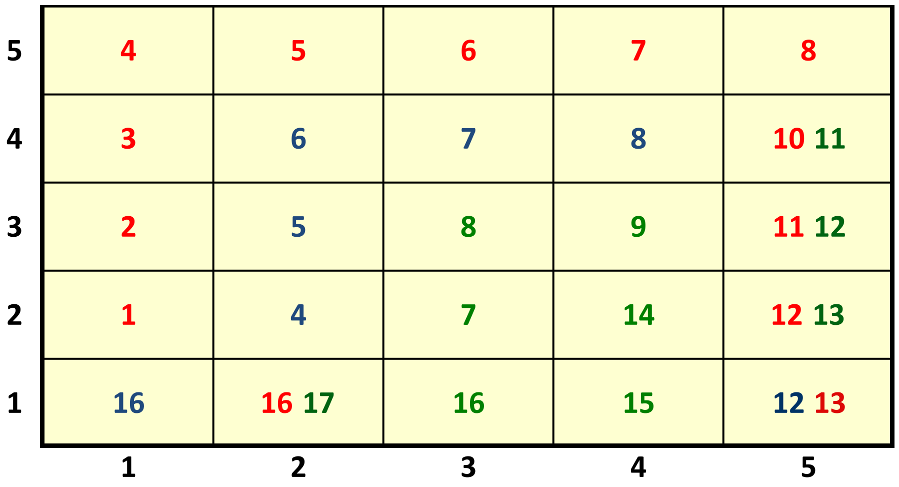


Figure 5.13: Progress cleaning of a five per five square floor using three cleaners and ignoring waiting conditions subtask.

of the problem it is stated that at each discrete time step, each cleaning robot cleans its current position assuming it is not critical. Moreover, the original CLEAN protocol allows inner cycles of the protocol (See Figure A.1), which means that a cleaner can execute parts of the CLEAN algorithm more than one time per time step. In Section 4.3.3 we explained how we deviated from the workflow of the original protocol due to the implicit synchronisation between cleaners in the local neighbourhood that the original CLEAN protocol in [31] uses without an explanation of the semantics behind it. Since the original protocol was highly unclear about the synchronisation mechanism between the cleaners, our decision was to model this implicit barrier syntonisation by introducing fixed delays using **await** statements. This way of synchronisation forces us to modify the original workflow by allowing subtasks

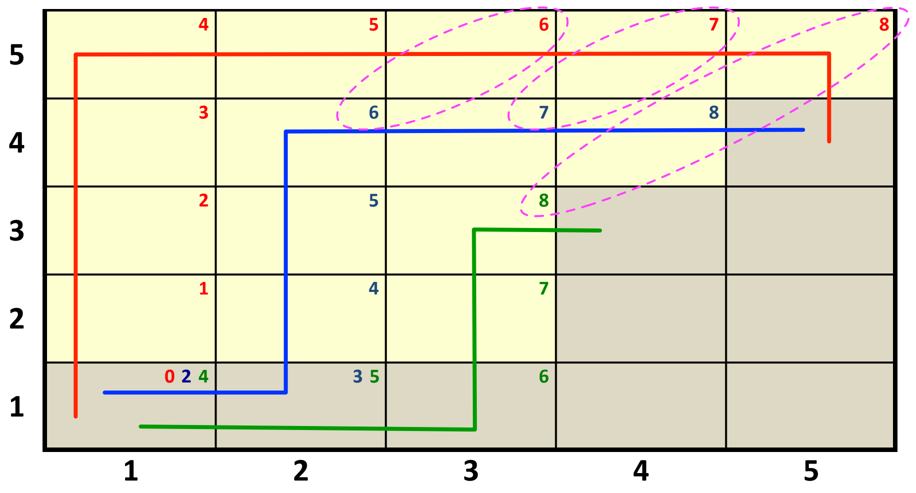


Figure 5.14: How synchronisation should work ideally.

to be executed at most one time per time step.

Figure 5.14 depicts an ideally expected behaviour of the cleaners until time step 8 with a more suitable implicit synchronisation. In this figure at time step 6, cleaner RED made an iteration of the CLEAN protocol and decided that it needed to wait for cleaner BLUE, cleaner BLUE executed its cycle of the CLEAN protocol and decided to clean position $(2, 4)$ and move to position $(3, 4)$. At this point cleaner RED again tried an inner cycle of the CLEAN protocol and decided that it could clean position $(3, 5)$ (according to conditions in Table 5.1, cleaners that move in the current time step are ignored in the first four cases while calculating waiting conditions) and move to position $(4, 5)$. Observe that all this could happened in the same time step. A similar sequence of decisions could happened at time step 7. At time step 8, cleaner RED waits for cleaner BLUE and cleaner BLUE waits for cleaner GREEN. Cleaner GREEN can clean position $(3, 3)$ and move to position $(4, 3)$. After that cleaner RED and BLUE could try again to clean and move, in the second iteration cleaner BLUE could proceed to clean position $(4, 4)$ and move to position $(5, 4)$, and in a third iteration cleaner RED finally could success to clean position $(5, 5)$ and move to position $(5, 4)$, as before all these three iterations would occur in the same time step.

After explaining this ideally expected behaviour we see that the state "is there still enough time?" in Figure A.1 could be interpreted as *"is there still another cleaner in my local neighbourhood that did not move yet?"*. We believe that for modelling the original workflow of the CLEAN protocol with this new interpretation, we need another way of synchronisation. It is left as an open question: *how could we implement an implicit barrier synchronisation between cleaners in the local neighbourhood without compromising the main characteristics of swarm robotics such as the absence of a central controller and autonomy?*.

Our last question is related to *fault-tolerance*. Fault-tolerance is a design property that enables a system to continue its intended operation when part of the system fails. The cooperative cleaners case study as it is defined in Chapter 2, states that the cleaning robots have knowledge of only part of their environment. This means that the cleaning robots do not know how many robots in total are involved in cleaning the floor. Since there is no assumption about the number of robots, a possible decrease in this number due to some of them failing, will not affect the operation of the rest of robots. We could assume that the robots that stop due to some fail in their hardware will not send any signal and that the rest of robots, which are operating normally, could just ignore them. Under this assumption we believe that the current design of our model support fault-tolerance with respect to the goal of cleaning the floor. Obviously, the model in this case would not comply with the requirements of the CLEAN protocol that states that all robots end up at the initial position.

5.5 Discussion on Nondeterminism in the Model

The aim of this discussion is to evaluate the strength of the analysis of our model of the cooperative cleaners by means of simulations.

The cleaners in this case study use the CLEAN protocol for deciding their actions. Each subtask of the CLEAN protocol deterministically decides what would be the next action of the cleaner, for example, the next intended position is the rightmost neighbour of the current position, the decision of which cleaners are resting and which cleaner is moving depends on the time step of entering a position and on a priority function that is deterministic, and so forth. For this reason we conjecture that the protocol is in fact deterministic, given the priority ordering and the delayed staring time of the different cleaners. Consequently, from an initial state that includes a floor, an initial position and a number of cleaners, there will only be one possible final state.

Assuming that our model is deterministic gives us strong confidence in the concluding result we obtained from our analysis based on simulations.

Chapter 6

Conclusions

6.1 Summary of the Case Study

In this thesis we have modelled in Real-Time ABS and analysed the static version of the cooperative cleaners case study based on the description in [31].

Our current model of the cooperative cleaners problem uses an approximation of the original protocol. This model had been accomplished after a series of iterations of the model, where we first polished the abstractions as well as the user-defined datatypes in order to achieve simplicity and clarity, and later used simulations to analyse, understand and correct some of the misbehaviour related to the correct computation of the different functions, the correct manipulation of the data in the model, a number of safety properties that we checked by introducing **assert** statements in the model and finally to partially check the correct behaviour of the CLEAN protocol. But there are still misbehaviours in the model that we only manage to identify, understand, and propose some ideas in the direction of a possible solution.

In the rest of this section we would like to recapitulate the main achievements we obtain while modelling and analysing the cooperative cleaners case study.

6.1.1 Main Modelling Decisions

As explained in Chapter 4, the main modelling decisions we made while modelling this case study can be summarised as follows:

1. *Abstractions.* We decided to abstract the hardware of the cleaning robots by using method calls. In our solution we introduced a modelling entity that we called *environment*. This entity acts as a server that returns locally observable information to the cleaners depending on their position. In a real implementation, they would have obtained this information by using their own sensors. This entity also responds to the actions of the cleaners, for example cleaning a position, moving and so forth.

2. *User-defined datatypes and their associated functions.* It was part of our modelling decision

to use user-defined datatypes and associated functions to represent and manipulate the information of the floor and the cleaning robots in the environment entity as well as in the limited memory of each cleaner. We used simulations to validate the correctness of the functional part of the model.

It was also part of our modelling decision what data was interesting to record as monitoring information and how to represent it.

3. *Interpretation and implementation of informal concepts as functions.* It was part of our modelling work to interpret the implicit and informal definitions of concepts that were given in the original description of the cooperative cleaners problem and find them a suitable executable implementation in ABS. We used simulations to validate their correctness.
4. *Ambiguities of the CLEAN protocol.* The original definition of the CLEAN protocol in [31] contains a number of ambiguities and unclarities. It was part of our modelling decision how we interpreted these ambiguities and adapted them to our model. The main adaptations can be summarised as follows:
   (a) *Interpretation of the implicit synchronisation.* In our model of the cooperative cleaners, we modelled the implicit synchronisation between the cleaners by introducing **await** statements combined with timers, in this way we could guarantee that all the cleaners progress homogeneously through the protocol. Due to this way of synchronisation we restricted the execution of the CLEAN protocol to only one cycle per time step while the original protocol allows more than one cycle per time step (See Figure A.1). While analysing the simulations, we realise that this restriction introduced unnecessary delays reducing the performance of the concurrent cleaning and moreover we think that this restriction may be the cause of the livelock situation we explain in Section 5.2.
   (b) *Replacement of the "Check Near Completion of Mission" subtask.* We decided to replace this subtask by the less ambiguous one described in the SWEEP [6] protocol (See Appendix B).
   (c) *Adaptation of the "Calculate waiting dependencies" subtask.* We decided to adapt this subtask in order to keep the autonomy of the cleaning robots. This adaptation is described in Table 5.1.

6.1.2 Main Analysis Results

The main results we obtained while analysing this case study using simulations can be summarised as follows:

1. We have validated that the manipulation of user-defined datatypes is correct.
2. We have validated that the implementation of the concepts given in the problem description is adequate.
3. We have observed that the recording of the monitoring information was consistent.

4. We have observed that the model satisfies the considered safety properties.

In addition, as explained in Chapter 5, we used the monitoring information recorded in the cleaners and in the environment to manually analyse the collective behaviour of the cleaners. The main results of this manual analysis can be summarised as follows:

1. The goal of cleaning a dirty square floor is achieved as expected when using one cleaner.
2. The goal of cleaning a dirty square floor with two cleaners is achieved, but with poor performance.
3. The goal of cleaning a dirty square floor with three cleaners is not achieved due to the presence of livelock.

Finally we believe that the reasons for a poor performance as well the cause of the livelock was the way we implemented the implicit barrier synchronisation between the cleaners by introducing delays. We suggest as a possible solution to find a suitable implicit synchronisation between cleaners in the local neighbourhood which is more tuned to the original protocol but at the same time keep the autonomy and decentralisation of the cleaners. We believe such a solution could improve the performance and hopefully remove the livelock.

6.2 Results of this Thesis

In Section 1.1 of the introduction, we considered two questions that we wish to answer in this thesis.

The first question was:

> **1.** *How can Real-Time ABS be used to naturally model autonomous, decentralised and self-organised systems such as swarm robotics?*

To answer this question, we will review our experience while modelling the cooperative cleaners case study and comment on how similar techniques could be applied to other case studies that share similar characteristics.

Special hardware. The members of swarm robotics are robots similar to the cleaners in the cooperative cleaners case study. They interact with other robots and with an environment. For their interaction they use special hardware such as devices for receiving and sending data, devices for sensing, devices for signalling, devices for performing tasks, for example, cleaning an area, lifting an object, identifying a target, and so forth. In the cooperative cleaners case study we abstracted the behaviour of the special hardware by method calls between the robots and a special entity that we called *environment*. A similar solution could be used to model other special hardware in the swarm robotics domain.

Passive environment. The members of the swarm interact with an environment that they can observe, sense and modify using their special devices. The environment itself is passive like the floor in the cooperative cleaners case study. In our model of the cooperative cleaners we encapsulate and extend this passive environment with a more reactive one that was able to respond and change when interacting with the cleaners. A similar solution could be used to model other passive environments in the swarm robotics domain.

Implicit communication. In the swarm robotics domain the communication between the robots is typically implicit, either because there is no direct communication and the robots only sense and signal, or there is broadcast communication without knowledge of who are receiving the messages. In contrast in object-orientation, communication is point to point, which means that there is only one sender and one receiver, and the sender knows the identity of the receiver in advance. In the cooperative cleaners case study, the environment entity was used to abstract from this kind of communication by transforming it into point to point communication between the environment and the robots. A similar idea could be used to model implicit communication for other case studies in the swarm robotics domain.

State representation. In general when modelling both a passive environment and the limited internal memory of entities, it is a good idea to use the functional layer of ABS which includes user-defined datatypes and functions. For the cooperative cleaners case study, we used user-defined datatypes to represent the floor and all the shared information of the cleaners such as location, state, signals and so forth. We also used user-defined datatypes for the internal representation of the local neighbourhood that each cleaner robot could observe. We used functions to manipulate all these data structures.

Work flow. In general when modelling systems in Real-Time ABS, it is possible to represent the workflow of entities using the functional layer, the imperative layer or a combination of both layers. In the cooperative cleaners case study we decided to combine both layers because it seems more natural to represent the workflow of the cleaning robots in that way. Our experience also pointed out that the more we use the functional layer to abstract the behaviour, the faster the simulation become. A similar approach could be used to model the workflow of robots for other case studies in the swarm robotics domain.

Monitoring information. In general when modelling systems in Real-Time ABS, it is possible to record information for further analysis. The user-defined datatypes and the functional language can be used for this purpose. For the cooperative cleaners case study we recorded the path of a cleaner on the floor, the trace of how the workflow of a cleaner was executed and the trace of how the floor was cleaned. A similar technique could be used to record monitoring information in other case studies in the swarm robotics domain.

Synchronisation. Recall that in the swarm robotic domain, robots need to synchronise in a suitable way in order to coordinate their work and avoid to interrupt other robots' actions.

Synchronisation is directly related to the concept of concurrency. Synchronisation is in general a mechanism to ensure that concurrent actions do not lead to unwanted behaviour. Modelling a suitable synchronisation mechanism for a given concurrent problem represents a challenge on its own.

In conclusion, we experienced through the modelling of the cooperative cleaners case study that autonomous, decentralised and self-organised systems such swarm robotics can almost be naturally model by exploiting the benefits of Real-Time ABS such as its simple concurrent model, abstractions using its functional layer and its object-oriented imperative layer, but it is also required to have fairly advance modelling skills to overcome the challenges introduced by this that these kinds of systems.

The second question was:

2. *To what extent can the simulation tool of Real-Time ABS help to analyse the collective behaviour of such systems?*

To answer this question, we will review our experience while analysing the cooperative cleaners case study.

Simulations. In general when analysing models in Real-Time ABS, it is possible to use simulations. We applied this technique to analyse the cooperative cleaners case study and it has been useful in order to progressively check that functions compute correctly and that methods execute as expected. It has been also useful for checking how the whole system behaves. On the other hand, swarm robotics emphasises a large number of robots and in our experience while running simulations of the cooperative cleaners case study using the Maude backend we had problems to successfully execute a square floor of 40 per 40 with one and two cleaning robots.

It seems that Maude can not handle a large state space and with a simulation that includes a big floor and the big amount of data that both the environment and each cleaning robots store as monitoring information, at some point the simulation tool become unresponsive. One of the weaknesses of Maude is its lack of input and output capabilities, which means that all data must be stored in memory and can not be read and/or written progressively from/to files. If the simulation tool could support input/output, then the monitoring information could be progressively written in a file and the state space could be kept small. It is also true that we could run simulations without storing any monitoring information, but such simulations would not really give us much feedback about the behaviour of an individual entity or the behaviour of the whole system. If the simulations run without problems we would only observe termination and maybe the ending time, but if the simulation encounters for example a livelock or a deadlock, we definitely need to store more information in order to analyse when and most important why this is happening.

Analysis of monitoring information. In general when analysing models in Real-Time ABS, it is also possible to record information for further analysis. In the cooperative cleaners case study we stored information to give us some insights about, for example, how each cleaning robot was moving around the floor over time, how the CLEAN algorithm was executed for each time step, and how the floor was being cleaned over time. We did a manual analysis of this data. This analysis was crucial to understand the behaviour of the different cleaners and also to understand the overall behaviour of the system. It was also helpful to detect misbehaviours.

On the other hand we experienced that the resulting data after running a simulation in Maude was not user friendly. It required from our side to first do some reformatting of the raw data in order to get a more readable format. After that we manually organised the resulting data in order to get a visual representation. We managed to do this kind of analysis for small floors with a small number of cleaners, we find it very hard to scale this kind of analysis for big floors with a large number of robots. It would have been very useful to have a more readable format to begin with, because it would have saved us time when we needed to quickly understand the behaviour of the system and to detect problems after small changes in the model.

In conclusion, we experience through the analysis of the cooperative cleaners case study using the simulation tool of Real-Time ABS that the collective behaviour of autonomous, decentralised and self-organised systems such swarm robotics can be analysed using simulations, but this analysis is restricted to small scenarios and it requires reformatting and reorganisation of the obtained raw data.

6.3 Future Work

In terms of future work we first discuss possible changes to the model and to the protocol itself, and second, possible extensions to the analysis part of our work.

As possible changes to the protocol and its model, it would be interesting to explore other synchronisation mechanisms between the cleaning robots. Some of the options could be, for example, to organise the robots in groups or to handle the **await** statements in a more flexible way.

As a variation of the protocol, we could group the robots by their location and choose a coordinator randomly which will be in charge of the synchronisation of its group. This approach may affect the autonomy of the members of the swarm, but on the other hand, fault-tolerance could still be handled since any member could act as coordinator, and as soon as one cleaning robot remains alive, the goal would be reached.

Another option could be to change the time needed for synchronisation by exploiting the dense time domain of Real-Time ABS. This can be achieved by letting a synchronisation cycle to be a fraction of the remaining time inside the time step, leaving time for another cycle if needed. In this way there would always be time for more than one cycle of the CLEAN protocol. In this case, the **await** statements would use a relative suspension time depending on the time that is left to the next discrete time step instead of the fixed suspension time that our current model uses. However, recall that the original protocol was highly unclear about this point.

Another interesting way to extend the case study would be to model the dynamic version of the problem [5, 6] using the SWEEP protocol. The dynamic version of the problem involves the spreading of contamination across the floor, which in practice means that the shape of the dirty floor grows over time. In this version we can no longer assume that the dirty shape is a single connected component with no holes, because due to the spreading of contamination, the graph can suddenly contain holes. Also, we can no longer assume that the cleaning robots are always located on the boundary of the dirty graph because after the dirty shape of the floor has grown, the cleaning robots may be inside the dirty graph and not only on the boundary.

Finally, it would be also interesting to see if we could obtain other results by using tool extensions for ABS with stronger verification techniques, for example, the verification of behavioural requirements for all possible executions using KeY [18], resource cost analysis using COSTA [2], and simulations on the ABS Java backend [33] that can support input/output of files. Unfortunately these tools are until now only available for Core ABS and not yet extended to be used with Real-Time ABS.

The emergent behaviour of systems like swarm robotics is a challenge
for formal methods, both with respect to how these systems
should be modelled, how their properties should be described and
how they can be analysed!

Bibliography

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[2] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. COSTABS: a cost and termination analyzer for ABS. In Oleg Kiselyov and Simon Thompson, editors, *Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM'12)*, pages 151–154. ACM, 2012.

[3] Elvira Albert, Samir Genaim, Miguel Gómez-Zamalloa, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In Michael Butler and Wolfram Schulte, editors, *FM 2011*, volume 6664 of *Lecture Notes in Computer Science*, pages 353–368. Springer, June 2011.

[4] Yaniv Altshuler, Alfred M. Bruckstein, and Israel A. Wagner. Swarm robotics for a dynamic cleaning problem. In *Proc. Swarm Intelligence Symposium (SIS'05)*, pages 209–216. IEEE, 2005.

[5] Yaniv Altshuler, Vladimir Yanovski, Israel A. Wagner, and Alfred M. Bruckstein. Swarm intelligence - searchers, cleaners and hunters. In *Swarm Intelligent Systems*, volume 26 of *Studies in Computational Intelligence*, pages 93–132. Springer Berlin Heidelberg, 2006.

[6] Yaniv Altshuler, Vladimir Yanovski, Israel A Wagner, and Alfred M Bruckstein. Multi-agent cooperative cleaning of expanding domains. *International Journal of Robotics Research*, 30(8):1037–1071, 2011.

[7] R. Alur, J. Esposito, M. Kim, V. Kumar, and I. Lee. Formal modeling and analysis of hybrid systems: A case study in multi-robot coordination. In *FM'99 âFormal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 212–232. Springer Berlin Heidelberg, 1999.

[8] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[9] Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *Innovations in Systems and Software Engineering*, 9(1):29–43, 2013.

[10] Eric Bonabeau, David Corne, Joshua Knowles, and Riccardo Poli. Swarm intelligence theory: A snapshot of the state of the art. *Theoretical Computer Science*, 411(21):2081–2083, May 2010.

[11] Eric Bonabeau, David Corne, and Riccardo Poli. Swarm intelligence: the state of the art special issue of natural computing. *Natural Computing*, 9(3):655–657, 2010.

[12] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudolf Schlatte, and Peter Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In Marco Bernardo and Valérie Issarny, editors, *Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer, 2011.

[13] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. In Eelco Visser and Jaakko Järvi, editors, *Proc. Ninth International Conference on Generative Programming and Component Engineering, (GPCE'10)*, pages 13–22. ACM, 2010.

[14] Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th Intl. Symposium on Formal Methods for Components and Objects (FMCO'10)*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2012.

[15] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[16] Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *16th European Symposium on Programming, ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007.

[17] M Bernardine Dias and Anthony (Tony) Stentz. A market approach to multirobot coordination. Technical Report CMU-RI -TR-01-26, Robotics Institute, 2001.

[18] Crystal Chang Din, Johan Dovland, and Olaf Owe. Compositional reasoning about shared futures. In *Proc. International Conference on Software Engineering and Formal Methods (SEFM'12)*, volume 7504 of *Lecture Notes in Computer Science*, pages 94–108. Springer, 2012.

[19] Reiner Hähnle. The abstract behavioral specification language: A tutorial introduction. In *Formal Methods for Components and Objects*, volume 7866 of *Lecture Notes in Computer Science*, pages 1–37. Springer, 2013.

[20] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011.

[21] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, March 2007.

[22] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Dynamic resource reallocation between deployment components. In J. S. Dong and H. Zhu, editors, *Proc. International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 646–661. Springer, November 2010.

[23] Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Validating timed models of deployment components with parametric concurrency. In B. Beckert and C. Marché, editors, *Proc. International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*, volume 6528 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2011.

[24] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. A formal model of object mobility in resource-restricted deployment scenarios. In Farhad Arbab and Peter Ölveczky, editors, *Proc. 8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, volume 7253 of *Lecture Notes in Computer Science*, pages 187–204. Springer, 2012.

[25] Kiriakos Kiriakidis and Diana F. Gordon-Spears. Formal modeling and supervisory control of reconfigurable robot teams. In *Formal Approaches to Agent-Based Systems*, volume 2699 of *Lecture Notes in Computer Science*, pages 92–102. Springer Berlin Heidelberg, 2003.

[26] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[27] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[28] Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the Second International Workshop on Feature-Oriented Software Development, FOSD 2010*, pages 49–56. ACM, 2010.

[29] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In Theo D'Hondt, editor, *European Conference on Object-Oriented Programming (ECOOP 2010)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer, June 2010.

[30] Jüri Vain, Tanel Tammet, Alar Kuusik, and Silver Juurik. Towards scalable proofs of robot swarm dependability. In *Proc. of the 11th Biennial Baltic Electronics Conference*, pages 199–202. IEEE, 2008.

[31] Israel A. Wagner, Yaniv Altshuler, Vladimir Yanovski, and Alfred M. Bruckstein. Cooperative cleaners: A study in ant robotics. *International Journal of Robotics Research*, 27(1):127–151, 2008.

[32] Alan F.T. Winfield, Wenguo Liu, Julien Nembrini, and Alcherio Martinoli. Modelling a wireless connected swarm of mobile robots. *Swarm Intelligence*, 2:241–266, 2008.

[33] Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *STTT*, 14(5):567–588, 2012.

Appendix A

The Original CLEAN Protocol

Below is the original CLEAN algorithms as it is described in [31].

- **System Initialisation** – initialising the system at the beginning of the cleaners' operation.
    1. Arbitrarily choose the initial pivot position p_0 from all the positions in the boundary of F_0.
    2. Artificially mark p_0 as critical.
    3. Place all the cleaners on p_0.
    4. For($i = 1; i \leq k; i++$) do
        (a) Call Cleaner Reset for cleaner i.
        (b) Call CLEAN for cleaner i.
        (c) Wait two time steps.
    5. End **System Initialisation**.
- **Cleaner Reset** – resetting counters and flags.
    1. $resting = false$.
    2. $destination = null$.
    3. $cleaned\ in\ last\ tour = false$.
    4. $waiting = \emptyset$.
    5. End **Cleaner Reset**.
- **Priority**.
    1. Assume the cleaner moved from position (x_0, y_0) to position (x_1, y_1) then
        (a) $priority = 2(x_1 - x_0) + (y_1, y_0)$.
    2. End **Priority**.
- **Check Completion of Mission**.

```
1. If (($(x,y) = p_0$) and $(x,y)$ has no dirty neighbours then
   (a) If $(x,y)$ is dirty then
       i. Clean $(x,y)$.
   (b) **STOP**.
2. End **Check Completion of Mission**.
```

- **Check Near Completion of Mission** – identify scenarios in which there are too many agents blocking each other, preventing the cleaning of the last few dirty tiles.

```
1. If (this is the first time this function is called for this agent) then
   (a) Jump to 3.
2. If(($(x,y) = p_0$) and (this is the end of a tour) then
   (a) If ($cleaned\ in\ last\ tour = false$) then
       i. $resting = true$.
       ii. **STOP**.
   (b) Else
       i. $cleaned\ in\ last\ tour = false$.
3. End **Check Near Completion of Mission**.
```

- **CLEAN Protocol** – controls cleaner number *i* actions after Cleaner Reset and until the mission is completed.

```
1. **Check Completion of Mission**.
2. **Check Near Completion of Mission**.
3. Calculate destination of movement.
   (a) $destination = rightmost\ neighbour\ of\ (x,y)$.
4. Signalling the desired destination.
   (a) $destination\ signal\ bits = destination$.
5. Calculate resting dependencies– solves the agents clustering problem.
   (a) Let all cleaners in $(x,y)$ except agent $i$ be divided into the following four groups:
       – $A_1$: Agents signalling towards any direction different than destination.
       – $A_2$: Agents signalling towards destination which entered $(x,y)$ before agent $i$.
       – $A_3$: Agents signalling towards destination which entered $(x,y)$ after agent $i$.
       – $A_4$: Agents signalling towards destination which entered $(x,y)$ in the same time step as agent $i$.
   (b) Let group $A_4$ be divided into the following two groups:
       – $A_{4a}$: Agents with lower priority than of agent $i$.
       – $A_{4b}$: Agents with higher priority than of agent $i$.
   (c) $resting = false$.
   (d) If($A_2 \neq \emptyset$) or ($A_{4b} \neq \emptyset$) then
       i. $resting = true$.
       ii. If (current time step did not end yet) then jump to 3 else jump to 9.
```

6. Calculate waiting dependencies – takes care of cleaner synchronisation.
   (a) $waiting = \emptyset$.
   (b) If $(x-1,y) \in F_t$ and contains a non-resting agent which didn't move in the current time step yet then $waiting = waiting \cup \{left\}$.
   (c) If $(x,y-1) \in F_t$ and contains a non-resting agent which didn't move in the current time step yet then $waiting = waiting \cup \{down\}$.
   (d) If $(x-1,y-1) \in F_t$ and contains a non-resting agent which didn't move in the current time step yet then $waiting = waiting \cup \{left\text{-}down\}$.
   (e) If $(x+1,y-1) \in F_t$ and contains a non-resting agent which didn't move in the current time step yet then $waiting = waiting \cup \{right\text{-}down\}$.
   (f) If $destination = right$ and position $(x+1,y)$ contains a cleaner j , and $destination_j \neq left$, and there are no other cleaners delayed by the operating cleaner (i.e. position $(x-1,y)$ does not contain an agent l where $destination_l = right$ and position $(x,y+1)$ does not contain any cleaners and position $(x+1,y)$ does not contain a cleaner n where $destination_n = left$), then ($waiting = waiting \cup \{right\}$) and ($waiting_j = waiting_j \setminus \{left\}$).
   (g) If $destination = up$ and position $(x,y+1)$ contains a cleaner j , and $destination_j \neq down$, and there are no other cleaners delayed by the operating cleaner (i.e. position $(x,y-1)$ does not contain an agent l where $destination_l = up$ and position $(x+1,y)$ does not contain any cleaners and position $(x,y+1)$ does not contain a cleaner n where $destination_n = down$), then ($waiting = waiting \cup \{up\}$) and ($waiting_j = waiting_j \setminus \{down\}$).
   (h) If ($waiting \neq \emptyset$) then
      i. If (current time step did not end yet) then jump to 3 else jump to 9.
7. Decide whether to clean (x,y) (meaning: whether or not the position is critical).
   (a) If (x,y) has two dirty tiles in its 4 Neighbours which are not connected via a sequence of dirty tiles from its 8 Neighbours then
      i. (x,y) is critical and should not be cleaned.
   (b) Else
      i. If (x,y) still has other cleaners in it then Do not clean (x,y).
      ii. Else
         A. Clean (x,y).
         B. $cleaned\ in\ last\ tour = true$
8. Move to destination.
9. Wait until the current time step ends.
10. Return to 1.

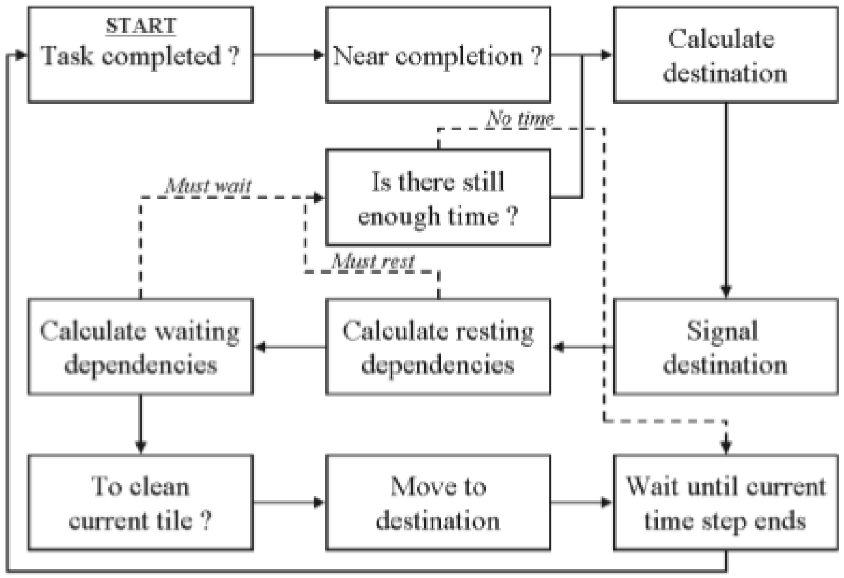


Figure A.1: A schematic flow chart of the CLEAN protocol. Extracted from [31].

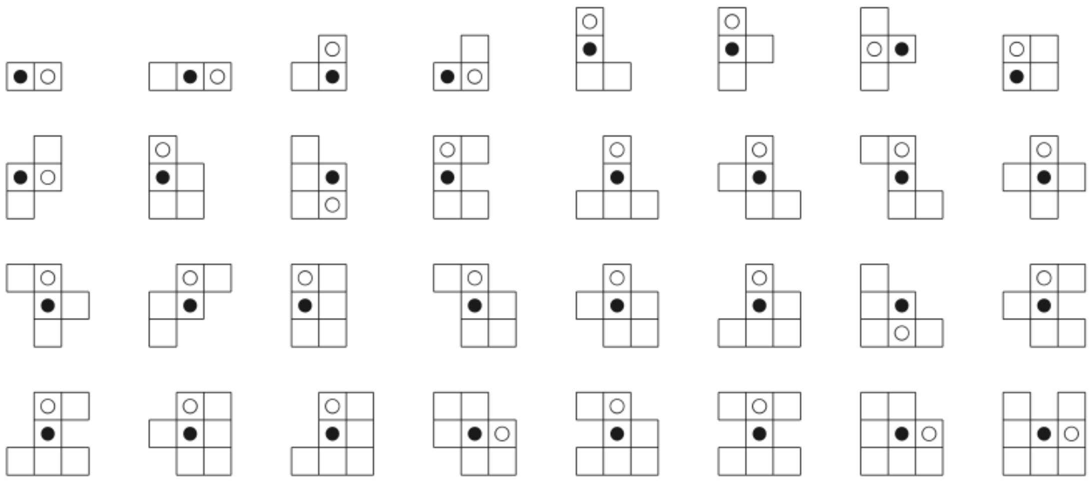


Figure A.2: First movement of a cleaner with initial position (x, y) should be decided according to this figure. Cleaner's initial position is marked as a filled circle while the intended next position is marked with an empty one. All configurations that do not appear in this figure can be obtained by using rotation and mirroring. Extracted from [31].

Appendix B

A Fragment of the Original SWEEP Protocol

Below is the original “Check Near Completion of Mission” subtask as it is described in [6]

- **Check Near Completion of Mission**.
  1. $near\ completion = false$
  2. If (each of the contaminated neighbours of (x, y) contains at least one agent) then
     (a) $near\ completion = true$
  3. If (each of the contaminated neighbours of (x, y) satisfy *near completion*) then
     (a) Clean (x, y)
     (b) **STOP**.
  4. $saturated\ perimeter = false$
  5. If (((x, y) is in the boundary of F_t) and (both (x, y) and all its non-critical neighbours in the boundary of F_t contain at least two agents)) then
     (a) $saturated\ perimeter = true$
  6. If (((x, y) is in the boundary of F_t) and (both (x, y) and all its non-critical neighbours in the boundary of F_t have *saturated perimeter*)) then
     (a) Ignore resting commands for this time step
  7. End **Check Near Completion of Mission**.

Appendix C

The Cooperative Cleaners Full Specification Code in Real-Time ABS

```
module CoopCleaners;

export *;

type Pos = Pair<Int, Int>;  // (x,y)
type Graph = Set<Pos>;
type PosSet = Set<Pos>;

data Status = Resting | Active | Stop;
//status, time of last movement, priority
type Cleaners =  Map<String,Triple<Status,Time,Int>>;
type CleanersPerPos = Map <Pos,Cleaners>;
// name and signalling postion
type CleanerSignal= Map<String, Pos>;


/////////////////ANALYSIS//////////////////////////////
type CleanerPath = List<Pair<Time,Pos>>;
type FloorPath = List<Triple<Time,Pos,String >>;
type CLEANPath = List<Triple<Time,String,Pos>>;
type FloorProgress = List<Pair<Time,Int>>;
/////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////


//////////////////SET FUNCTIONS//////////////
// compare two sets
def Bool compareSet<A>(Set<A> a, Set<A> b) =
if ~(size(a) == size(b)) then False
else if emptySet(a) && emptySet(b)
then True
else let (A elem) = take(a) in
 if contains(b, elem) then compareSet(remove(a, elem), remove(b, elem))
 else False;

def Bool isSubset<A>(Set<A> set1, Set<A> set2)=
case set1 { EmptySet => True;
            Insert(e,rest) =>
                case contains(set2,e) {
                    True => isSubset(rest,set2);
                    False => False;};};
```

```
////////////////////////////////////////////////
/////////////MAP FUNCTIONS////////////////////////////
def List<A> keysList<A, B>(Map<A, B> map) =
  case map {
    EmptyMap => Nil ;
    InsertAssoc(Pair(a, _), tail) => Cons(a, keysList(tail));
  };

///////////////////////////////////////////////////////////////
/////////////////floor////////////////////////////
def Graph insertRow(Int x, Int limX, Int y, Graph floor)=
case x {
    limX => Insert(Pair(x,y),floor);
    _ => insertRow(x+1,limX,y,Insert(Pair(x,y),floor));
};

def Graph makeFloor(Int x, Int limX, Int y, Int limY, Graph floor)=
case y {
    limY => insertRow(x,limX,y,floor);
    _ => makeFloor(x,limX,y+1,limY,insertRow(x,limX,y,floor));
};
///////////////////////////////////////////////////////////


def Bool posIsDirty(Pos p,Graph g) =  contains(g, p);

def Pos up(Pos p) = case p {Pair(x,y) => Pair(x,y+1);};
def Pos upleft(Pos p) = case p {Pair(x,y) => Pair(x-1,y+1);};
def Pos upright(Pos p) = case p {Pair(x,y) => Pair(x+1,y+1);};
def Pos left(Pos p) = case p  {Pair(x,y) => Pair(x-1,y);};
def Pos right(Pos p) = case p {Pair(x,y) => Pair(x+1,y);};
def Pos down(Pos p) = case p  {Pair(x,y) => Pair(x,y-1);};
def Pos downleft(Pos p) = case p {Pair(x,y) => Pair(x-1,y-1);};
def Pos downright(Pos p) = case p {Pair(x,y) => Pair(x+1,y-1);};

def PosSet dirtySet(PosSet s, Graph g)=
   case s {
       EmptySet => EmptySet ;
       Insert(p,tail) =>
          case posIsDirty(p,g) {
              True => Insert(p,dirtySet(tail,g));
              False => dirtySet(tail,g);
          };
   };


def PosSet eightNbr(Pos p, Graph g) =
  let (Pos u)=up(p) in
  let (Pos ul)=upleft(p) in
  let (Pos ur)=upright(p) in
  let (Pos l)=left(p) in
  let (Pos r)=right(p) in
  let (Pos d)=down(p) in
  let (Pos dl)=downleft(p) in
  let (Pos dr)=downright(p) in
  dirtySet(set[u,ul,ur,l,r,d,dl,dr],g);


def PosSet fourNbr(Pos p, Graph g)=
   let (Pos u)=up(p) in
   let (Pos d)=down(p) in
   let (Pos l)=left(p) in
   let (Pos r)=right(p) in
   dirtySet(set[u,d,l,r],g);

def PosSet fourPos(Pos p, Graph g)=
   let (Pos u)=up(p) in
   let (Pos d)=down(p) in
   let (Pos l)=left(p) in
   let (Pos r)=right(p) in
   set[u,d,l,r];

def Bool neighborsDirty(Graph g) = ~ emptySet(g);

////////////////////////////////////////////////////////////////////////////////////
//////////////CHECK TASK NEAR COMPLETION////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////


def Bool checkEightNbsCond(Pos p, Graph g, CleanersPerPos c) =
   let (PosSet eightnbr) =  eightNbr(p, g) in
       checkOneCleanerPerPos(Insert(p,eightnbr),g,c);

def Bool checkOneCleanerPerPos(PosSet nbr,Graph g, CleanersPerPos c) =
   case nbr {
       EmptySet => True;
       Insert(h,t) => checkEightNbr(eightNbr(h, g),c) &&
                      checkOneCleanerPerPos(t, g,  c);
   };

def Bool checkEightNbr(PosSet eightnbr, CleanersPerPos c) =
    case eightnbr {
        EmptySet => True;
        Insert(h,t) =>
            case lookupDefault(c, h, EmptyMap) {
                EmptyMap => False;
                _ => True && checkEightNbr(t, c);
                };
            };

def Bool checkSaturatedPerimeter(Pos p, Graph g, CleanersPerPos c) =
    let (PosSet eightnbr) =  eightNbr(p, g) in
       checkTwoCleanerPerCritPos(Insert(p,eightnbr),g,c);

def Bool checkTwoCleanerPerCritPos(PosSet nbr,Graph g, CleanersPerPos c) =
   case nbr {
       EmptySet => True;
       Insert(h,t) => checkPerimeter(eightNbr(h, g),c,g) &&
                      checkTwoCleanerPerCritPos(t, g,  c);
   };

def Bool checkPerimeter(PosSet eightnbr, CleanersPerPos c, Graph g) =
    case eightnbr {
        EmptySet => True;
        Insert(h,t) =>
          if (posIsInBorder(h,g) && ~isCritical(h,g)) then
            case lookupDefault(c, h, EmptyMap) {
                EmptyMap => False;
                InsertAssoc(cl,EmptyMap) => False;
                _ => True && checkPerimeter(t, c,g);
            }
          else
             checkPerimeter(t, c,g);
    };


////////////////////////////////////////////////////////////////////////////////////
//////////////CALCULATE DESTINATION////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////

def Bool posIsInBorder(Pos p, Graph g) = size(eightNbr(p,g))<8 ;

def Pos tmove(Pos pp, Pos p)=
  let (Pos l)=left(p) in
  let (Pos u)=up(p) in
  let (Pos r)=right(p) in
  let (Pos d)=down(p) in
  case pp { l => up(p);
            u => right(p);
            r => down(p);
            d => left(p);};

def Pos destination(Pos pp,Pos pos, Graph fullNeighbors, Pos ip) =
    case neighborsDirty(eightNbr(pos,fullNeighbors)) {
       False => case (contains(fourPos(pos, fullNeighbors),ip)){
               True => ip;
               False => pos;};
       True => case posIsDirty(tmove(pp, pos),fullNeighbors) &&
                    posIsInBorder(tmove(pp,pos),fullNeighbors) {
               True => tmove(pp,pos);
               False => destination(tmove(pp,pos),pos, fullNeighbors,ip);};};


/////////////////////////////////////////////////////////////
/////////////////Priority//////////////////////////////
def Int priority(Pos pos0 ,Pos pos1) =
   case pos1 {
      Pair(x1,y1) => case pos0 {
         Pair(x0,y0) => 2*(x1-x0) + (y1-y0);};};


/////////////////////////////////////////////////////////
////////////sensing activities///////////////////////
////////////////////////////////////////////////////////

/////////////LOCAL FLOOR//////////////////////////
def Graph localGraph(Pos p, Graph g) =
   let (Graph cs) = eightNbr(p, g) in
   let (Graph uls) =  eightNbr(upleft(p), g) in
   let (Graph urs) = eightNbr(upright(p), g) in
   let (Graph dls) =  eightNbr(downleft(p), g) in
   let (Graph drs) = eightNbr(downright(p), g) in
   union(cs, union(uls,union(urs,union(dls,drs))));

//////////////LOCAL CLEANERS PER DESTINATION///////////////////
def CleanersPerPos simplyfyCleaners(CleanersPerPos c) =
  case c {  EmptyMap => EmptyMap;
            InsertAssoc(head,tail) =>
              case snd(head)== EmptyMap {
                 True => simplyfyCleaners(tail);
                 False => InsertAssoc(head,simplyfyCleaners(tail));
              };
  };

def CleanersPerPos localCleaners(Pos p,CleanersPerPos cXp ) =
  let (Pair<Pos,Cleaners> ulul)=Pair(upleft(upleft(p)),
      lookupDefault(cXp,upleft(upleft(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> ulu)=Pair(upleft(up(p)),
      lookupDefault(cXp,upleft(up(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> uu)=Pair(up(up(p)),
      lookupDefault(cXp,up(up(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> uru)=Pair(upright(up(p)),
      lookupDefault(cXp,upright(up(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> urur)=Pair(upright(upright(p)),
      lookupDefault(cXp,upright(upright(p)), EmptyMap)) in

  let (Pair<Pos,Cleaners> ull)=Pair(upleft(left(p)),
      lookupDefault(cXp,upleft(left(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> ul)=Pair(upleft(p),
      lookupDefault(cXp,upleft(p), EmptyMap)) in
  let (Pair<Pos,Cleaners> u)=Pair(up(p),
      lookupDefault(cXp,up(p), EmptyMap)) in
  let (Pair<Pos,Cleaners> ur)=Pair(upright(p),
      lookupDefault(cXp,upright(p), EmptyMap)) in
  let (Pair<Pos,Cleaners> urr)=Pair(upright(right(p)),
      lookupDefault(cXp,upright(right(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> ll)=Pair(left(left(p)),
      lookupDefault(cXp,left(left(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> l)=Pair(left(p),
      lookupDefault(cXp,left(p), EmptyMap)) in
  let (Pair<Pos,Cleaners> c)=Pair(p,
      lookupDefault(cXp,p, EmptyMap)) in
  let (Pair<Pos,Cleaners> r)=Pair(right(p),
      lookupDefault(cXp,right(p), EmptyMap)) in
  let (Pair<Pos,Cleaners> rr)=Pair(right(right(p)),
      lookupDefault(cXp,right(right(p)), EmptyMap)) in

  let (Pair<Pos,Cleaners> dll)=Pair(downleft(left(p)),
      lookupDefault(cXp,downleft(left(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> dl)=Pair(downleft(p),
      lookupDefault(cXp,downleft(p), EmptyMap)) in
  let (Pair<Pos,Cleaners> d)=Pair(down(p),
      lookupDefault(cXp,down(p), EmptyMap)) in
  let (Pair<Pos,Cleaners> dr)=Pair(downright(p),
      lookupDefault(cXp,downright(p), EmptyMap)) in
  let (Pair<Pos,Cleaners> drr)=Pair(downright(right(p)),
      lookupDefault(cXp,downright(right(p)), EmptyMap)) in

  let (Pair<Pos,Cleaners> dldl)=Pair(downleft(downleft(p)),
      lookupDefault(cXp,downleft(downleft(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> dld)=Pair(downleft(down(p)),
      lookupDefault(cXp,downleft(down(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> dd)=Pair(down(down(p)),
      lookupDefault(cXp,down(down(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> drd)=Pair(downright(down(p)),
      lookupDefault(cXp,downright(down(p)), EmptyMap)) in
  let (Pair<Pos,Cleaners> drdr)=Pair(downright(downright(p)),
      lookupDefault(cXp,downright(downright(p)), EmptyMap)) in
  simplyfyCleaners(map[ulul,ulu,uu,uru,urur,ull,ul,u,ur,urr,ll,l,
                     c,r,rr,dll,dl,d,dr,drr,dldl,dld,dd,drd,drdr]);


////////////// LOCAL SIGNALS ////////////////////////////////////////////////////////////

def CleanerSignal localSignals(  Set<String> cleaners, CleanerSignal fullSignalsMap) =
  case cleaners {
       EmptySet => EmptyMap;
       Insert(n,tail) =>
           case (lookup(fullSignalsMap,n)== Nothing) {
               True => localSignals(tail, fullSignalsMap);
               False => InsertAssoc(Pair(n,fromJust(lookup(fullSignalsMap,n))),
                                          localSignals(tail, fullSignalsMap));
            };
    };

def Set<String> listCleaners(CleanersPerPos cleanersMap) =
   case cleanersMap {
       EmptyMap => EmptySet;
       InsertAssoc(h,t) => union(keys(snd(h)),listCleaners(t));
    };



////////////////////////////////////////////////////////
////////////Calculating Resting Dependecies////////////
////////////////////////////////////////////////////

def Cleaners  removeCleanerName(Cleaners  map, String name) =
    case map  {
        EmptyMap => EmptyMap;
        InsertAssoc(Pair(n, p), tail) =>
              case n==name {
                  True => tail;
                  False => InsertAssoc(Pair(n, p),removeCleanerName(tail, name));
              };
    };


def List<String> removingSignalDifPos(CleanerSignal localSignals,
                                      List<String> cleanersSamePos, Pos signalPos) =
     case cleanersSamePos  {
        Nil => Nil;
        Cons(head, tail) =>
            case (lookup(localSignals,head) == Just(signalPos)){
                True => Cons(head,removingSignalDifPos(localSignals,tail,signalPos)) ;
                False => removingSignalDifPos(localSignals,tail,signalPos);
            };
     };

def List<String> enterSameTimeAsCleaner(Cleaners cleaners,
                                        List<String> cleanersSamePosSameSignal,
                                        Time lastMove) =
     case cleanersSamePosSameSignal  {
        Nil => Nil;
        Cons(head, tail) =>
             case (lookup(cleaners,head) != Nothing ) {
                 True =>
                    case (lastMove==sndT(fromJust(lookup(cleaners,head)))){
                        False => enterSameTimeAsCleaner(cleaners,tail,lastMove);
                        True => Cons(head, enterSameTimeAsCleaner(cleaners,tail,lastMove));
                    };
                 False => enterSameTimeAsCleaner(cleaners,tail,lastMove);
             };
     };


def Bool enterBeforeCleaner(Cleaners cleaners, List<String> cleanersSamePosSameSignal,
                            Time lastMove) =
     case cleanersSamePosSameSignal  {
        Nil => False;
        Cons(head, tail) =>
             case (lookup(cleaners,head) != Nothing ) {
                 True =>
                    case (timeLessThan(sndT(fromJust(lookup(cleaners,head))),lastMove)){
                         True => True;
                         False => enterBeforeCleaner(cleaners,tail,lastMove);};
                 False => enterBeforeCleaner(cleaners,tail,lastMove);
             };
     };


def Bool isHigherPriority(Cleaners cleaners,
                          List<String> cleanersSamePosSameSignalSameTime, Int priority) =
     case cleanersSamePosSameSignalSameTime  {
        Nil => False;
        Cons(head, tail) =>
           case (lookup(cleaners,head) != Nothing ) {
              True => case (trd(fromJust(lookup(cleaners,head)))>priority){
                  True => True;
                  False => isHigherPriority(cleaners,tail,priority);};
              False => isHigherPriority(cleaners,tail,priority);
           };
     };

//////////////////////////////////////////////////////////////////////
///////////////CALCULATING WAITING DEPENDECIES////////////////////////
/////////////////////////////////////////////////////////////////////

def Cleaners filterActive(Cleaners map) =
    case map {
        EmptyMap => EmptyMap;
        InsertAssoc(Pair(n, tr),tail) =>
                case fstT(tr) {
                    Active =>  InsertAssoc(Pair(n, tr),filterActive(tail));
                    _ => filterActive(tail);

                };
    };

def Cleaners filterNotMoveThisTime(Cleaners  map, Time t) =
    case map {
        EmptyMap => EmptyMap;
        InsertAssoc(Pair(n, p),tail) =>
                case timeLessThan(sndT(p),t) {
                    True =>  InsertAssoc(Pair(n, p),filterNotMoveThisTime(tail,t));
                    False => filterNotMoveThisTime(tail,t);
                };
    };

def Bool checkNotDestination( Cleaners cleaners, Pos nextPos, CleanerSignal signals) =
   case cleaners {
       EmptyMap => False;
       InsertAssoc(Pair(n,_),tail) =>
            case (lookup(signals,n)== Just(nextPos))  {
                 False =>  checkNotDestination(tail, nextPos, signals);
                 True => True;
            };

    };

def Bool thereIsCleanerNotSignalingTo( Cleaners cleaners,CleanerSignal signals, Pos pos) =
  case cleaners {
       EmptyMap => False;
       InsertAssoc(Pair(n,p),tail) =>
       case (lookup(signals,n)!= Just(pos)) {
               True => True;
               False => thereIsCleanerNotSignalingTo(tail,signals,pos) ;
            };
    };

def Bool thereIsCleanerSignalingTo( Cleaners  cleaners,CleanerSignal signals, Pos pos) =
  case cleaners {
       EmptyMap => False;
       InsertAssoc(Pair(n,p),tail) =>
       case (lookup(signals,n)== Just(pos)) {
               True => True;
               False => thereIsCleanerSignalingTo(tail,signals,pos) ;
            };
    };

def Bool thereIsNoCleanerWithSignalTo(Cleaners cleaners,CleanerSignal signals, Pos pos) =
  case cleaners {
       EmptyMap => True;
       InsertAssoc(Pair(n,p),tail) =>
       case (lookup(signals,n)== Just(pos)) {
               True => False;
               False => thereIsCleanerNotSignalingTo(tail,signals,pos) ;
            };
    };

//////////////////////////////////////////////////////////////
//////////////CRITICAL FUNCTIONS/////////////////////////////
/////////////////////////////////////////////////////////////

def Bool isCritical(Pos p, Graph g)= ~isSingleConnectedComponnet(eightNbr(p,g));


//////////////////////////////////////////////////////////////////////////////
////////////////////////analysis/////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////////

def Bool isFloorDirty(Graph graph) =  ~ emptySet(graph) ;
```

```
////////////////////////CONNECTIVITY/////////////////////

def Graph dfs(Graph dfsNodes, Graph visited, Graph dirtyFloor) =
    case  dfsNodes  {
        EmptySet => visited;
        Insert(n,newDfsNodes) => case contains(visited,n) {
            True => dfs(newDfsNodes,visited,dirtyFloor);
            False => dfs(union(newDfsNodes, fourNbr(n, dirtyFloor)),
                         insertElement(visited,n ),dirtyFloor);};};

def Bool isSingleConnectedComponnet(Graph dirtyFloor) =
     case emptySet(dirtyFloor)  {
        True => True;
        False => compareSet(dirtyFloor,dfs(set[take(dirtyFloor)],EmptySet,dirtyFloor));
    };


///////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////// ENVIRONMENT ////////////////////////////////
///////////////////////////////////////////////////////////////////////////////////////

interface Environment {

   Graph getLocalDirtyFloor(Pos pos);
   CleanersPerPos getLocalCleaners(Pos pos);
   CleanerSignal getLocalSignals(Set<String> cleaners);

   Unit cleanTile(Pos pos, String name, Time timeStep);
   Graph getDirtyFloor();
   Unit initSignal(String name, Pos pos);
   Unit updateDestinationSignal(String name, Pos nextPos);
   Unit updateStatus(String name, Pos pos, Status s, Time t,Int pr );
   Unit updatePos(String name,  Pos pos, Pos nextPos, Status s, Time t,Int pr );
   Unit stopSignalDestination(String name);
}

class EnvironmentImp(Graph g) implements Environment {

  Graph dirtyFloor = g;
  CleanersPerPos cleanersXPos = EmptyMap;
  CleanerSignal signalsMap = EmptyMap;
  Bool floorComplete = False;

  ///////////////////////////////////////////////
  /////////////ANALYSIS/////////////////////////
  //////////////////////////////////////////////
  FloorPath  floorPath = Nil;
  FloorProgress floorProgress = Nil ;
  //////////////////////////////////////////////

  Graph getLocalDirtyFloor(Pos pos) {return localGraph(pos, dirtyFloor)  ;}

  CleanersPerPos getLocalCleaners(Pos pos) {
      return  simplyfyCleaners(localCleaners(pos,cleanersXPos));}

  CleanerSignal getLocalSignals(Set<String> cleaners) {
      return localSignals( cleaners, signalsMap);}

  Unit cleanTile(Pos pos, String name, Time timeStep) {
      Graph newFloor = remove(dirtyFloor, pos);
      //////////analysis//////////////////
      assert isSingleConnectedComponnet(newFloor);
      dirtyFloor = newFloor;
      floorPath = Cons(Triple(timeStep, pos, name), floorPath);
  }

  Unit run(){
      while (dirtyFloor != EmptySet){
           floorProgress =  Cons(Pair(now(), size(dirtyFloor)),floorProgress);
           await duration(5,5);
      }
  }

  Graph getDirtyFloor(){return dirtyFloor;}

  Unit initSignal(String name, Pos pos) {
      Cleaners  tmpMap = put(lookupDefault(cleanersXPos, pos, EmptyMap),
                             name,Triple(Active,now(), 0));
      cleanersXPos =  put(cleanersXPos, pos, tmpMap);
  }

  Unit updateDestinationSignal(String name, Pos nextPos) {
      signalsMap = put(signalsMap,name,nextPos);

  }

 Unit stopSignalDestination(String name) {
     signalsMap = removeKey(signalsMap,name);
 }

 Unit updateStatus(String name, Pos pos, Status s,Time t,Int pr ) {
      Cleaners tmpMap = put(lookupDefault(cleanersXPos, pos, EmptyMap),
                            name,Triple(s,t,pr));
      cleanersXPos =  put(cleanersXPos, pos, tmpMap);
 }

 Unit updatePos(String name, Pos pos, Pos nextPos, Status s, Time t, Int pr )
 {
      Cleaners rmvTmpMap =  removeKey(lookupDefault(cleanersXPos, pos, EmptyMap),name);
      Cleaners  addTmpMap =  put(lookupDefault(cleanersXPos, nextPos, EmptyMap),
                                 name,Triple(s,t,pr));
      cleanersXPos = put(cleanersXPos, pos, rmvTmpMap);
      cleanersXPos = simplyfyCleaners(put(cleanersXPos, nextPos, addTmpMap));

 }
}

/////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////// CLEANER  ////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////////

interface Cleaner {
    Unit reset(Pos p0, Environment e, Bool cLEANPathOn, Bool cleanerPathOn);
    CLEANPath getCLEANPath();
    CleanerPath  getCleanerPath();
    Unit stopped();
}


class CleanerImp(String name) implements Cleaner {
  Bool resting = False;
  Pos nextPos = Pair(-999,-999);
  Bool cleanInLastTour = False;
  PosSet waitingSet = EmptySet;
  Bool nearCompletion = False;
  Bool saturatedPerimeter = False;


  Time timeStep =  Time(0);
  Graph localDirtyFloor = EmptySet;
  CleanersPerPos localCleaners = EmptyMap;
  CleanerSignal localSignals = EmptyMap;
  Pos initPos = Pair(-999,-999);
  Pos pos = Pair(-999,-999);
  Pos prevPos = Pair(-999,-999);
  Int priority = 0;
  Time lastTimeMove = Time(0);
  Bool stop = False;
  Environment env;

  ///// ANALYSIS VARIABLES/////////////
  CleanerPath  cleanerPath = Nil;
  CLEANPath  pCLEANPath = Nil;
  Bool onCLEANPath = False;
  Bool onCleanerPath = False;

  CLEANPath  getCLEANPath() { return pCLEANPath;  }

  CleanerPath getCleanerPath(){return cleanerPath ;}
 ///////////////////////////////////


 Unit stopped(){ await stop; }

  Unit reset(Pos p0, Environment e, Bool cLEANPathOn, Bool cleanerPathOn){
      resting = False;
      nextPos = Pair(-999,-999);
      cleanInLastTour = True;
      waitingSet = EmptySet;
      nearCompletion = False;
      saturatedPerimeter = False;

      timeStep =  now();
      localDirtyFloor = EmptySet;
      localCleaners = EmptyMap;
      localSignals = EmptyMap;
      initPos = p0;
      pos = p0;
      prevPos = left(pos);
      priority = priority(prevPos,pos);
      lastTimeMove = now();
      env = e;
      stop = False;

      ///// ANALYSIS VARIABLES/////////////
      onCleanerPath  = cleanerPathOn;
      onCLEANPath =  cLEANPathOn;
      pCLEANPath  = Nil;
      cleanerPath = Nil;
      //////////////////////////////////

      Fut<Unit> f = env!initSignal(name, pos); f.get;
      this!checkTaskCompleted();
  }


  Unit sense(){
      ////////////////// sensing from the environment //////////////////////////////////
      Fut<Graph> f1 = env!getLocalDirtyFloor(pos);
      localDirtyFloor = f1.get;
      Fut<CleanersPerPos> f2 = env!getLocalCleaners(pos);
      localCleaners = f2.get;
      Fut<CleanerSignal> f3 = env!getLocalSignals(listCleaners(localCleaners));
      localSignals = f3.get;
  }

  ///////////////////////////////////////////////////////////
  ///////////// checkTaskCompleted ////////////////////////////////////
 ///////////////////////////////////////////////////////////

  Unit checkTaskCompleted(){
      timeStep = now();
      if (onCLEANPath) { pCLEANPath = Cons(Triple(now(),"CTC",pos),pCLEANPath);}
      if (onCleanerPath ) {cleanerPath = Cons(Pair(timeStep,pos),cleanerPath);}
      this.sense();
      Bool isNeigborhoodDirty =  neighborsDirty(eightNbr(pos,localDirtyFloor));
      if (pos == initPos && isNeigborhoodDirty==False )
      {
          if (posIsDirty(pos,localDirtyFloor))
          {
              Fut<Unit> f = env!cleanTile(pos,name,timeStep); f.get;
          }
          stop = True;
          Fut<Unit> f = env!updateStatus(name, pos, Stop, timeStep,priority); f.get;
      }
      else  {
          this!checkTaskNearCompletion();

      }
  }

  ///////////////////////////////////////////////////////////
  /////////////checkTaskNearCompletion /////////////////////////////////////
 ///////////////////////////////////////////////////////////


 Unit checkTaskNearCompletion(){

      if (onCLEANPath) { pCLEANPath  = Cons(Triple(now(),"CNC",pos),pCLEANPath) ;}
      if  (pos == initPos && checkEightNbsCond(pos,localDirtyFloor,localCleaners)){
          if (posIsDirty(pos,localDirtyFloor))
          {
              Fut<Unit> f = env!cleanTile(pos,name,timeStep); f.get;
          }
          stop = True;
          Fut<Unit> f = env!updateStatus(name, pos, Stop, timeStep,priority); f.get;
      }
      saturatedPerimeter = False;
      if (checkSaturatedPerimeter(pos, localDirtyFloor,localCleaners)) {
          saturatedPerimeter = True;
      }
     if (~stop) { this!calcDestination();}
  }

  ///////////////////////////////////////////////////////////
  ///////////// calcDestination ////////////////////////////////////
 ///////////////////////////////////////////////////////////

  Unit calcDestination(){
      if (onCLEANPath) { pCLEANPath  = Cons(Triple(now(),"CD",pos),pCLEANPath ) ;}
      resting = False;
      Fut<Unit> f = env!updateStatus(name, pos, Active, lastTimeMove, priority); f.get;

      nextPos = destination(prevPos, pos, localDirtyFloor,initPos);
      this!signalDestination();
  }
  ///////////////////////////////////////////////////////////
  ////////////  signalDestination  /////////////////////////////////////
 ///////////////////////////////////////////////////////////
  Unit signalDestination(){
      if (onCLEANPath) { pCLEANPath  = Cons(Triple(now(),"SD",pos),pCLEANPath) ;}

      Fut<Unit> f = env!updateDestinationSignal(name, nextPos); f.get;
      this!calcRestingDependecies();
  }

  ///////////////////////////////////////////////////////////
  ///////////// calcRestingDependecies //////////////////////
 ///////////////////////////////////////////////////////////
  Unit calcRestingDependecies(){
      await duration(1/4,1/4);
      if (onCLEANPath) { pCLEANPath  = Cons(Triple(now(),"CRD",pos),pCLEANPath ) ;}
      this.sense();
      if (~saturatedPerimeter) {
          Cleaners cleanersMapWithSamePos = lookupDefault(localCleaners,pos, EmptyMap);
          cleanersMapWithSamePos = removeCleanerName(cleanersMapWithSamePos, name);
          List<String> cleanersWithSamePos = keysList(cleanersMapWithSamePos);
          List<String> cleanersWithSamePosSameSignal =
                         removingSignalDifPos(localSignals, cleanersWithSamePos, nextPos);
          List<String> cleanersWithSamePosSameSignalSameTime  =
                         enterSameTimeAsCleaner(cleanersMapWithSamePos,
                                                   cleanersWithSamePosSameSignal,
                                                   lastTimeMove);

          if (enterBeforeCleaner(cleanersMapWithSamePos,cleanersWithSamePosSameSignal,
                                 lastTimeMove)
              ||isHigherPriority(cleanersMapWithSamePos,
                                 cleanersWithSamePosSameSignalSameTime ,priority)){
              resting = True;
              Fut<Unit> f = env!updateStatus(name, pos, Resting, lastTimeMove, priority);
              f.get;
              await duration(3/4,3/4);
              this!checkTaskCompleted();
          }
          else  {
              await duration(1/4,1/4);
              this!calcWaitingSet();
          }
      }
      else {
          await duration(1/4,1/4);
          this!calcWaitingSet();
     }
  }

  ////////////////////////////////////////////////////////////
  /////////////calcWaitingSet ///////////////////////////////////
 ////////////////////////////////////////////////////////////

  Unit calcWaitingSet(){
      if (onCLEANPath) { pCLEANPath  = Cons(Triple(now(),"CWD",pos),pCLEANPath ) ;}
      waitingSet = EmptySet;
      this.sense();

      //case a
      Cleaners tmpMap =  filterNotMoveThisTime(filterActive(lookupDefault(
                                  localCleaners,left(pos), EmptyMap)),timeStep);
      if (posIsDirty(left(pos),localDirtyFloor)  && tmpMap !=  EmptyMap) {
          waitingSet = insertElement(waitingSet,left(pos));
      }
      //case b
      tmpMap =  filterNotMoveThisTime(filterActive(lookupDefault(
                                  localCleaners ,down(pos), EmptyMap)),timeStep);
      if (posIsDirty(down(pos),localDirtyFloor)  && tmpMap !=  EmptyMap) {
          waitingSet = insertElement(waitingSet,down(pos));
      }
      //case c
      tmpMap =  filterNotMoveThisTime(filterActive(lookupDefault(
                                   localCleaners ,downleft(pos), EmptyMap)),timeStep);
      if (posIsDirty(downleft(pos),localDirtyFloor)  && tmpMap !=  EmptyMap) {
          waitingSet = insertElement(waitingSet,downleft(pos));
      }
      //case d
      tmpMap =  filterNotMoveThisTime(filterActive(lookupDefault(
                                   localCleaners ,downright(pos), EmptyMap)),timeStep);
      if (posIsDirty(downright(pos),localDirtyFloor)  && tmpMap !=  EmptyMap) {
          waitingSet = insertElement(waitingSet,downright(pos));
      }
      // CASE e
      if (nextPos == right(pos) ) {
          Cleaners tmpMapRight = filterActive(lookupDefault(localCleaners,right(pos),
                                              EmptyMap));
          Cleaners tmpMapLeft = filterActive(lookupDefault(localCleaners,left(pos),
                                             EmptyMap));
          Cleaners tmpMapUp = filterActive(lookupDefault(localCleaners,up(pos),
                                           EmptyMap));
          //(x+1,y) contains an agent j and detination(j) != left
          if (thereIsCleanerNotSignalingTo(tmpMapRight,localSignals, pos)) {
              // there are not other agents delayed by the operating agent
              //(x-1,y) does not contain any agent l where destination(l) == right
              //(x,y+1) does not contain any agents
              //(x+1,y) does not contain any agent n where destination(n)== left
              if (thereIsNoCleanerWithSignalTo(tmpMapLeft,localSignals, pos) &&
                  tmpMapUp==EmptyMap &&
                  thereIsNoCleanerWithSignalTo(tmpMapRight,localSignals, pos) ) {
                  waitingSet = insertElement(waitingSet,right(pos));
              }
          }
      }
          // CASE f
      if (nextPos != left(pos) ) {
          tmpMap =  filterActive(lookupDefault(localCleaners,pos,
                                 EmptyMap));
          Cleaners tmpMapLeft = filterActive(lookupDefault(localCleaners,left(pos),
                                              EmptyMap));
          Cleaners tmpMap2Left = filterActive(lookupDefault(localCleaners,left(left(pos)),
                                              EmptyMap));
          Cleaners tmpMapUpLeft = filterActive(lookupDefault(localCleaners,upleft(pos),
                                               EmptyMap));
          //(x-1,y) contains an agent y and detination(y) == right
          if (thereIsCleanerSignalingTo(tmpMapLeft,localSignals, pos)) {
              // there are not other agents delayed
              //(x-2,y) does not contain any agent l where destination(l) == right
              //(x-1,y+1) does not contain any agents
              //(x,y) does not contain any agent n where destination(n)== left
              if (thereIsNoCleanerWithSignalTo(tmpMap2Left,localSignals, left(pos)) &&
                  tmpMapUpLeft==EmptyMap &&
                  thereIsNoCleanerWithSignalTo(tmpMap,localSignals, left(pos))) {
                  waitingSet = remove(waitingSet,left(pos));
              }
          }
      }

          // CASE g
      if (nextPos == up(pos) ) {
          Cleaners tmpMapUp = filterActive(lookupDefault(localCleaners,up(pos),
                                           EmptyMap));
          Cleaners tmpMapDown = filterActive(lookupDefault(localCleaners,down(pos),
                                             EmptyMap));
          Cleaners tmpMapRight = filterActive(lookupDefault(localCleaners,right(pos),
                                              EmptyMap));
          //(x,y+1) contains an agent j and detination(j) != left
          if (thereIsCleanerNotSignalingTo(tmpMapUp,localSignals, pos)) {
              // there are not other agents delayed by the operating agent
              //(x,y-1) does not contain any agent l where destination(l) == up
              //(x+1,y) does not contain any agents
              //(x,y+1) does not contain any agent n where destination(n)== down
              if (thereIsNoCleanerWithSignalTo(tmpMapDown,localSignals, pos) &&
                  tmpMapRight==EmptyMap &&
                  thereIsNoCleanerWithSignalTo(tmpMapUp,localSignals, pos)) {
                  waitingSet = insertElement(waitingSet,up(pos));
              }
          }
      }

      // CASE h
      if (nextPos != down(pos) ) {
          tmpMap =  filterActive(lookupDefault(localCleaners,pos,
                                 EmptyMap));
          Cleaners tmpMapDown = filterActive(lookupDefault(localCleaners,down(pos),
                                             EmptyMap));
```

```
          Cleaners tmpMap2Down = filterActive(lookupDefault(localCleaners,down(down(pos)),
                                                    EmptyMap));
          Cleaners tmpMapDownRight =filterActive(lookupDefault(localCleaners,downright(pos),
                                                        EmptyMap));
          //(x,y-1) contains an agent y and detination(y) == up
          if (thereIsCleanerSignalingTo(tmpMapDown,localSignals, pos)) {
              // there are not other agents delayed
              //(x,y-2) does not contain any agent l where destination(l) == up
              //(x+1,y-1) does not contain any agents
              //(x,y) does not contain any agent n where destination(n)== down
              if (thereIsNoCleanerWithSignalTo(tmpMap2Down,localSignals, down(pos)) &&
                  tmpMapDownRight==EmptyMap &&
                  thereIsNoCleanerWithSignalTo(tmpMap,localSignals, down(pos))) {
                  waitingSet = remove(waitingSet,down(pos));
              }
          }
      }
          //checking status of waiting set
      if (~emptySet(waitingSet))  {
          await duration(1/2,1/2);
          this!checkTaskCompleted();
      }
      else{
          this.mayCleanCurrentPos();
      }
  }

  ////////////////////////////////////////////////////////////
  ////////////mayCleanCurrentPos /////////////////////////
////////////////////////////////////////////////////////////

  Unit mayCleanCurrentPos() {
   if (onCLEANPath) { pCLEANPath  = Cons(Triple(now(),"MCCP",pos),pCLEANPath ) ;}
   if (pos != initPos) {
      if ( ~isCritical(pos, localDirtyFloor)  &&
          (posIsDirty(nextPos,localDirtyFloor) || nextPos==initPos) &&
          removeKey(lookupDefault(localCleaners,pos,EmptyMap),name) ==EmptyMap) {
          Fut<Unit> f = env!cleanTile(pos,name,timeStep); f.get;
          cleanInLastTour = True;
      }
      else {
          cleanInLastTour = False;
      }
   }

   if (posIsDirty(nextPos,localDirtyFloor) || nextPos == initPos){
       this.moveToDestination();
       }
   else{
      await duration(1/2,1/2);
      this!checkTaskCompleted();
       }
  }

  ////////////////////////////////////////////////////////////
  ////////////moveToDestination /////////////////////////////////////
////////////////////////////////////////////////////////////

  Unit moveToDestination(){
      if (onCLEANPath) { pCLEANPath  = Cons(Triple(now(),"MD",pos),pCLEANPath) ;}
      lastTimeMove = timeStep;
      priority = priority(pos,nextPos);
      Fut<Unit> f = env!updatePos(name, pos, nextPos, Active, lastTimeMove,priority);
      f.get;
      f = env!stopSignalDestination(name); f.get;
      prevPos = pos;
      pos = nextPos;
      await duration(1/2,1/2);
      this!checkTaskCompleted();
  }

}

//**************************************************************
//***************************main*****************************
//**************************************************************


{

//////////////// Example Shape with 2 cleaners //////////////////////////////////

//8 |              D  D  D
//7 |              D  D  D
//6 |              D  D  D
//5 |D  D          D  D  D
//4 |D  D  D  D    D  D  D
//3 |D  D  D  D  D D  D  D
//2 |D  D  D  D    D  D  D
//1 |D  D          D  D
//  _______________________
//   1  2  3  4  5  6  7  8

  /*     Graph initialDirtyFloor = makeFloor(1,2, 1, 5,
                                 makeFloor(3,4, 2, 4,
                                 makeFloor(5,5, 3, 3,
                                 makeFloor(6,7, 1, 8,
                                 makeFloor(8,8, 2, 8, EmptySet)))));
       assert isSingleConnectedComponnet(initialDirtyFloor);
       Environment env = new cog EnvironmentImp(initialDirtyFloor);
       Pos p0 = Pair(2,5);
       Cleaner c1 = new cog CleanerImp("RED");
       Cleaner c2 = new cog CleanerImp("BLUE");

       c1!reset(p0, env,True,True);
       Fut<Unit> finishC1 = c1!stopped();
       await duration(2,2);
       c2!reset(p0, env,True,True);
       Fut<Unit> finishC2 = c2!stopped();

       await finishC1? ;
       await finishC2? ;

       Graph finalDirtyFloor  = await env!getDirtyFloor();
       Bool isTheInitialFloorDirty = isFloorDirty(initialDirtyFloor);
       Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor);  */

/////////////////////////////////////////////////////////////////

//////////////////////F5x5_1c/////////////////////////////////

       Graph initialDirtyFloor = makeFloor(1,5, 1, 5,EmptySet);
       assert isSingleConnectedComponnet(initialDirtyFloor);
       Environment env = new cog EnvironmentImp(initialDirtyFloor);
       Pos p0 = Pair(1,1);
       Cleaner c1 = new cog CleanerImp("RED");
       c1!reset(p0, env,True,True);
       Fut<Unit> finishC1 = c1!stopped();
       await finishC1? ;
       Graph finalDirtyFloor  = await env!getDirtyFloor();
       Bool isTheInitialFloorDirty = isFloorDirty(initialDirtyFloor);
       Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor);
////////////////////////////////////////////////////////////////

//////////////////////F5x5_2c/////////////////////////////////

  /*     Graph initialDirtyFloor = makeFloor(1,5, 1, 5,EmptySet);
       assert isSingleConnectedComponnet(initialDirtyFloor);
       Environment env = new cog EnvironmentImp(initialDirtyFloor);
       Pos p0 = Pair(1,1);
       Cleaner c1 = new cog CleanerImp("RED");
       Cleaner c2 = new cog CleanerImp("BLUE");

       c1!reset(p0, env,True,True);
       Fut<Unit> finishC1 = c1!stopped();
       await duration(2,2);
       c2!reset(p0, env,True,True);
       Fut<Unit> finishC2 = c2!stopped();

       await finishC1? ;
       await finishC2? ;

       Graph finalDirtyFloor  = await env!getDirtyFloor();
       Bool isTheInitialFloorDirty = isFloorDirty(initialDirtyFloor);
       Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor); */
///////////////////////////////////////////////////////////////////

///////////////////////F5x5_3c/////////////////////////////////

     /*  Graph initialDirtyFloor = makeFloor(1,5, 1, 5,EmptySet);
       assert isSingleConnectedComponnet(initialDirtyFloor);
       Environment env = new cog EnvironmentImp(initialDirtyFloor);
       Pos p0 = Pair(1,1);
       Cleaner c1 = new cog CleanerImp("RED");
       Cleaner c2 = new cog CleanerImp("BLUE");
       Cleaner c3 = new cog CleanerImp("GREEN");

       c1!reset(p0, env,True,True);
       Fut<Unit> finishC1 = c1!stopped();
       await duration(2,2);
       c2!reset(p0, env,True,True);
       Fut<Unit> finishC2 = c2!stopped();
       await duration(2,2);
       c3!reset(p0, env,True,True);
       Fut<Unit> finishC3 = c3!stopped();

       await finishC1? ;
       await finishC2? ;
       await finishC3? ;

       Graph finalDirtyFloor  = await env!getDirtyFloor();
       Bool isTheInitialFloorDirty = isFloorDirty(initialDirtyFloor);
       Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor);  */

///////////////////////////////////////////////////////////////////

///////////////////////F10x10_1c/////////////////////////////////
    /*   Graph initialDirtyFloor = makeFloor(1,10, 1, 10,EmptySet);
       assert isSingleConnectedComponnet(initialDirtyFloor);
       Environment env = new cog EnvironmentImp(initialDirtyFloor);
       Pos p0 = Pair(1,1);
       Cleaner c1 = new cog CleanerImp("RED");

       c1!reset(p0, env,True,True);
       Fut<Unit> finishC1 = c1!stopped();

       await finishC1? ;

       Graph finalDirtyFloor  = await env!getDirtyFloor();
       Bool isTheInitialFloorDirty = isFloorDirty(initialDirtyFloor);
       Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor); */
///////////////////////////////////////////////////////////////////

///////////////////////F10x10_2c/////////////////////////////////
 /*      Graph initialFloor = makeFloor(1,10, 1, 10,EmptySet);
       assert isSingleConnectedComponnet(initialDirtyFloor);
       Environment env = new cog EnvironmentImp(initialDirtyFloor);
       Pos p0 = Pair(1,1);
       Cleaner c1 = new cog CleanerImp("RED");
       Cleaner c2 = new cog CleanerImp("BLUE");

       c1!reset(p0, env,True,True);
       Fut<Unit> finishC1 = c1!stopped();
       await duration(2,2);
       c2!reset(p0, env,True,True);
       Fut<Unit> finishC2 = c2!stopped();

       await finishC1? ;
       await finishC2? ;

       Graph finalDirtyFloor  = await env!getDirtyFloor();
       Bool isTheInitialFloorDirty = isFloorDirty(initialDirtyFloor);
       Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor); */
//////////////////////////////////////////////////////////////////

//////////////////////F10x10_3c/////////////////////////////////
  /*     Graph initialDirtyFloor = makeFloor(1,10, 1, 10,EmptySet);
       assert isSingleConnectedComponnet(initialDirtyFloor);
       Environment env = new cog EnvironmentImp(initialDirtyFloor);
       Pos p0 = Pair(1,1);
       Cleaner c1 = new cog CleanerImp("RED");
       Cleaner c2 = new cog CleanerImp("BLUE");
       Cleaner c3 = new cog CleanerImp("GREEN");

       c1!reset(p0, env,True,True);
       Fut<Unit> finishC1 = c1!stopped();
       await duration(2,2);
       c2!reset(p0, env,True,True);
       Fut<Unit> finishC2 = c2!stopped();
       await duration(2,2);
       c3!reset(p0, env,True,True);
       Fut<Unit> finishC3 = c3!stopped();

       await finishC1? ;
       await finishC2? ;
       await finishC3? ;

       Graph finalDirtyFloor  = await env!getDirtyFloor();
       Bool isTheInitialFloorDirty = isFloorDirty(initialDirtyFloor);
       Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor);    */
//////////////////////////////////////////////////////////////////

//////////////////////F20x20_1c/////////////////////////////////

   /*    Graph initialDirtyFloor = makeFloor(1,20, 1, 20,EmptySet);
       assert isSingleConnectedComponnet(initialDirtyFloor);
       Environment env = new cog EnvironmentImp(initialDirtyFloor);
       Pos p0 = Pair(1,1);
       Cleaner c1 = new cog CleanerImp("RED");

       c1!reset(p0, env,True,True);
       Fut<Unit> finishC1 = c1!stopped();

       await finishC1? ;

       Graph finalDirtyFloor  = await env!getDirtyFloor();
       Bool isTheInitialFloorDirty = isFloorDirty(initialDirtyFloor);
       Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor); */

//////////////////////////////////////////////////////////////////

//////////////////////F20x20_2c/////////////////////////////////
    /*   Graph initialDirtyFloor = makeFloor(1,20, 1, 20,EmptySet);
       assert isSingleConnectedComponnet(initialDirtyFloor);
       Environment env = new cog EnvironmentImp(initialDirtyFloor);
       Pos p0 = Pair(1,1);
       Cleaner c1 = new cog CleanerImp("RED");
       Cleaner c2 = new cog CleanerImp("BLUE");

       c1!reset(p0, env,True,True);
       Fut<Unit> finishC1 = c1!stopped();
       await duration(2,2);
       c2!reset(p0, env,True,True);
       Fut<Unit> finishC2 = c2!stopped();

       await finishC1? ;
       await finishC2? ;

       Graph finalDirtyFloor  = await env!getDirtyFloor();
       Bool isTheInitialFloorDirty = isFloorDirty(initialDirtyFloor);
       Bool isTheFinalFloorDirty = isFloorDirty(finalDirtyFloor);  */
///////////////////////////////////////////////////////////////////

}

// Local Variables:
// abs-use-timed-interpreter: t
// abs-clock-limit: 50
// End:
```