Chapter 12
Assurance of Agent Systems: What Role Should Formal Verification Play?

M. Winikoff

Abstract In this chapter we consider the broader issue of gaining assurance that an agent system will behave appropriately when it is deployed. We ask to what extent this problem is addressed by existing research into formal verification. We identify a range of issues with existing work which leads us to conclude that, broadly speaking, verification approaches on their own are too narrowly focussed. We argue that a shift in direction is needed, and outline some possibilities for such a shift in direction.

M. Winikoff
University of Otago, New Zealand e-mail: michael.winikoff@otago.ac.nz

12.1 Introduction

A key issue in the deployment of multi-agent systems is being able to obtain assurance that the system will behave correctly. Before deploying a system, we need to convince those who will rely on the system (or those who will be liable/responsible if it fails) that the system will, in fact, work. Traditionally, this assurance is done through testing. However, it is generally accepted that adaptive systems exhibit a wider and more complex range of behaviours, making testing harder. For example, Munroe *et al.* [320, Section 3.7.2] say that:

> ... validation through extensive tests was mandatory ... However, the task proved challenging for several reasons. First, agent-based systems explore realms of behaviour outside people's expectations and often yield surprises ...

The lack of good ways of obtaining assurance that an agent system will behave correctly is seen to be a significant obstacle to industrial adoption. For instance, one of the four obstacles to widespread adoption of agent technology in manufacturing noted by Hall *et al.* [204] is

> Can the aggregate behavior of the agent-based system be guaranteed to meet all the system requirements?

Similarly, Pěchouček and Mařík [349, Page 413] note that[1]:

> Although the agent system performed very well in all the tests, to release the system for production would require testing all the steel recipes with all possible configurations of cooling boxes.

Before going further we need to briefly introduce some terminology. The term "**assurance**" refers to a process that aims to obtain confidence that (in this case) an agent system will behave appropriately. The term is used in a broad (and somewhat imprecise) sense. Where there is a clear specification (which is not always the case!) then we can use the two standard terms "verification" and "validation". Verification in this context refers to checking whether software meets its specification, and validation refers to checking whether the specification meets the user's requirements. There are a range of techniques for performing verification, including *formal verification* where mathematical reasoning techniques are used to formally establish a relationship between software and a formal mathematical specification. In some cases the formal specification is the whole software specification, in other cases it may be certain key properties, e.g. freedom from deadlock.

Unfortunately, the state of the art in assuring the correct behaviour of agent systems is still somewhat limited, and, perhaps surprisingly, there is not a large amount

[1] On the other hand, for another application they note that [349, Page 407]: "*Even though this negotiation process has not been theoretically proved for cycles' avoidance* [sic], *practical experiments have validated its operation*"

of existing and ongoing work. It is only relatively recently that testing began to receive significant attention from the agents community (e.g. [157,199,322,323,447]), and existing work on verification (typically using model checking; see section 12.2) is still rather limited in the size of systems that can be verified.

This chapter's key contribution is to look at the broad issue of obtaining assurance, and to ask to what extent this need is likely to be met by (the eventual outcomes of) current research on agent system verification. Put more concisely, if assuring correct behaviour is the problem, what role does formal verification have in the solution? We identify a number of issues which lead us to conclude that existing approaches are only *part* of the solution. We then suggest some alternative directions for investigation that aim to find out how to use existing work on testing and on verification as part of an assurance solution for agent systems.

This chapter doesn't question *whether* formal methods should be used, since it is clear that they have a role to play [436]. Rather, this chapter questions the traditional (narrow) view of verification typically assumed in agent verification research, and argues that, to be used as part of a broadly applicable approach for obtaining assurance, the scope of verification needs to be broadened, and certain assumptions need to be reconsidered. One possible broadly applicable approach for obtaining assurance is presented, and one key aspect (combining testing and proving) is discussed in more detail.

This chapter is structured as follows. We briefly review existing work on agent system verification (section 12.2), then introduce a simple case study (section 12.3) and prove that it is correct (section 12.4) before proceeding to discuss various issues (section 12.5) and considering how some of these issues manifest themselves in the case study (section 12.6). Section 12.7 proposes a new approach for assurance of agent systems, and section 12.8 elaborates on a key part of the approach: combining testing and proving techniques. We conclude in section 12.9.

12.2 Existing Work

In this section we briefly set the context by reviewing some existing work on verification of agent systems. Note that we focus in this section on verification, and do not describe in detail any of the work on testing agent systems which, roughly speaking, covers either support for running tests on agent systems (e.g. [157,199]), and/or ways of generating test cases (based on design models [447], ontologies [322], or using evolutionary techniques where soft-goals are rendered as quality functions that are used to guide the evolution of good test cases [323]). Note that this section is not intended to be a comprehensive survey, but merely to give a flavour of the recent work that has been done in the area. For a more comprehensive overview of the area we refer the reader to other chapters in this book.

All of the work discussed below considers, in some form, the problem of establishing beyond doubt (i.e. through proof or exhaustive analysis) that an agent program[2] $\mathcal{P}$ meets a specification $\mathcal{S}$ (typically given in temporal logic, or an extension thereof). Much (but not all) of the existing work focues on model checking.

An early piece of work on verification was by Wooldridge *et al.* [441]. It introduced a simple imperative language (MABLE) and a specification language ($\mathcal{MORA}$, a simplified form of Wooldridge's $\mathcal{LORA}$). The $\mathcal{MORA}$ notation combines temporal logic and dynamic logic, and adds modalities for beliefs, desires and intentions. Each agent's program is translated into a Promela process, and claims about the system (in $\mathcal{MORA}$) are translated and checked using the SPIN model checker.

Bordini and colleagues [68, 73] extended this work by adopting as their agent programming language an agent-oriented programming language: AgentSpeak(F), a subset of Rao's AgentSpeak(L) limited to be finite. A subsequent paper [72] introduced a slicing algorithm which eliminates parts of the agent program that are not relevant to the property being checked, thus reducing the state space and the effort required to check the property (time required reduced by around 25% for the two examples they considered).

More recently, their work has shifted to support a wider range of agent-oriented programming languages, by translating to a common underlying abstract language, AIL, which is then translated to Java, and checked using a variant of the Java PathFinder tool[3] called AJPF [140]. Compared with using JPF, AJPF shows significant efficiency improvements [67]. However, the programs being verified are still "toy" programs, e.g. a six-line contract net example with three agents.

Another strand of work is that of Lomuscio and colleagues (e.g. [355]) which focuses on verifying so-called interpreted systems where, roughly speaking, an agent is modelled as a finite state machine. The specification logic is temporal logic augmented with a knowledge operator. Some recent work [168] is interesting in that it explicitly considers injecting faults, and looks at a range of specifications (in CTL) and what they mean in terms of the requirement on the system to be able to recover from faults. For example. a (CTL) specification of the form $AG(injected \rightarrow EF\phi)$ (where ϕ is the desired property) requires the system to have the possibility of eventually recovering from a fault, but doesn't require that it always do so, whereas $AG(injected \rightarrow AF\phi)$ requires that the system always (eventually) recover from an injected fault. While a promising direction in model checking research, this particular work is still somewhat preliminary: they model checked a single bit transmission protocol.

All of the work discussed so far in this section has used model checking. However, there is also some work that uses theorem proving. The work of Shapiro *et al.* [395] defined the Cognitive Agent Specification Language (CASL), and proved properties of ConGolog programs by translating to PVS, a typed higher-order logic with available tool support. They were able to verify properties of a simple meeting scheduler.

[2] In some work this is a single agent program, in other work this is a collection of agent programs, as well as a model of the system's environment

[3] http://javapathfinder.sourceforge.net/

More recent work includes that of Alechina *et al.* [6–8] which also uses tool-supported theorem proving. They use a simple language (SimpleAPL), and translate SimpleAPL programs (along with the starting state of the agent) into Propositional Dynamic Logic (PDL). Safety and liveness properties can then be verified using an existing PDL theorem prover. A key contribution of Alechina *et al.* is the ability to model the agent's execution strategy, and prove properties that may rely on a given execution strategy (e.g. interleaved vs. non-interleaved execution of plans). The approach appears to have only been applied to toy programs (single agent, a couple of plans, and propositional beliefs).

Finally, another example of theorem proving is the work of Mermet *et al.* [312] which proves correctness using proof schemas. The agents are specified using Goal Decomposition Trees which specify how each goal is achieved by its sub-goals, for example, by a sequence of sub-goals. Unlike model checking, showing that a robot on a grid works correctly does not depend on the size of the grid. However, the proofs appear to be done by hand (tool support is mentioned as future work).

12.3 Case Study: A Waste Disposal Robot

Our discussion of issues will be made concrete and illustrated using a case study. Our case study scenario is a simple model of a waste disposal robot, and is inspired by examples used in a range of previous papers (e.g. [68, 312, 323, 357, 379]). Note that certain aspects of the case study are done in a way that could be improved (for instance not having a separate sense action); this was done in order to better allow a range of issues to be illustrated using a single case study.

The world consists of a grid of locations $\mathbb{L} = \{0 \ldots \mathrm{MAX}\} \times \{0 \ldots \mathrm{MAX}\}$ (for some value of $MAX \in \mathbb{N}$), with typical element $\ell \in \mathbb{L}$. Each location contains an amount of rubbish which is specified by the model $m : \mathbb{L} \to \mathbb{N}$, that is, for a given location ℓ the amount of rubbish at that location is $m(\ell) \in \mathbb{N}$. A certain subset of the locations $b \subseteq \mathbb{L}$ has bins for the disposal of rubbish (and it is assumed that these locations are known to the robot).

A robot's state consists of its position $p \in \mathbb{L}$, the amount of rubbish it is holding $h \in \mathbb{N}$, and its view of the environment $v : \mathbb{L} \to \mathbb{N} \cup \{\bot\}$, where $v(\ell) = \bot$ denotes that the robot does not know anything about the location ℓ (we assume in what follows that any numerical condition on $v(\ell)$ such as $v(\ell) > 0$ has an implicit additional condition that $v(\ell) \neq \bot$). The robot also has a fixed capacity for carrying rubbish, $c > 0$.

The overall goal of the robot is to eliminate all rubbish: $\forall \ell : v(\ell) = 0$. In fact, we also want the robot to not be holding rubbish, and so the robot's goal $\mathcal{G}$ is $h = 0 \wedge \forall \ell . v(\ell) = 0$.

The actions available to the robot are specified in Figure 12.1 and comprise moving, picking up and dropping rubbish, and sensing how much rubbish is located at its current location. Actions are specified using simultaneous assignment statements, rather than the more traditional post-conditions for two reasons: it is closer to an implementation, and it avoids needing to specify explicitly what things are left unchanged by an action.

Summary of model

$\mathbb{L} = \{0 \ldots MAX\} \times \{0 \ldots MAX\}$	locations	$p \in \mathbb{L}$	robot's position
$\ell \in \mathbb{L}$	location	$h \in \mathbb{N}$	rubbish held by robot
$m : \mathbb{L} \rightarrow \mathbb{N}$	environment model	$v : \mathbb{L} \rightarrow \mathbb{N} \cup \{\perp\}$	robot's (partial) view
$b \subseteq \mathbb{L}$	locations of bins	$c \in \mathbb{N}$	robot's carrying capacity

Actions

Action	Pre-condition	Effect (simultaneous assignment)
move(ℓ_{dest})	true	$p := \ell_{dest}$
pick()	$h < c$	$h := h + t$ $m(p) := m(p) - t$ $v(p) := v(p) - t$ where $t = \min(c - h, m(p))$
drop()	$p \in b$	$h := 0$
sense()	true	$v(p) := m(p)$

Robot Program

(Notation: *event* : *condition* ← *planbody*)

1a $+!clean : v(p) = \perp \leftarrow$ sense() ; $!clean$
1b $+!clean : h = 0 \wedge \forall \ell . v(\ell) = 0 \leftarrow$ stop (do nothing)
1c $+!clean : h = 0 \wedge v(p) = 0 \wedge \ell \in \mathbb{L} \wedge v(\ell) > 0 \leftarrow$ move(ℓ) ; $!clean$
1c' $+!clean : h = 0 \wedge v(p) = 0 \wedge \forall \ell' . v(\ell') \in \{0, \perp\} \wedge \ell \in \mathbb{L} \wedge v(\ell) = \perp \leftarrow$ move(ℓ) ; $!clean$
2 $+!clean : h = 0 \wedge v(p) > 0 \leftarrow$ pick() ; $!clean$
3 $+!clean : h > 0 \wedge p \notin b \wedge \ell \in b \leftarrow$ move(ℓ) ; $!clean$
4 $+!clean : h > 0 \wedge p \in b \leftarrow$ drop() ; $!clean$

Fig. 12.1 Summary of model, actions, and robot program

The robot's program could be specified in a wide range of notations, ranging from simple to quite complex. We use the AgentSpeak(L) notation [357] to specify the robot's program, although a simpler notation such as condition-action rules would have sufficed. The agent program captures the following cases (see Figure 12.1 for details).

1. If the robot is not carrying anything and there is no rubbish at its location, then explore[4]

[4] In fact, there are a few sub-cases here: if the robot does not know how much rubbish is at its current location it should perform a sense action, if the robot knows about the location of rubbish,

2. If the robot is not carrying any rubbish and there is rubbish at the current location, then pick it up
3. If the robot is carrying rubbish and it is not at a bin, then move to a bin
4. If the robot is carrying rubbish and is at a bin, then drop the rubbish

This basic robot can be extended and improved in various ways, such as improving its efficiency. One issue is that the robot picks up rubbish, and then proceeds to a bin immediately, even though it may be able to carry more rubbish. Our first optimisation ("opt1") is to have the robot continue to collect rubbish until it is full. This is done by modifying the condition $h = 0$ in plans 1c and 2 to $h < c$. A second efficiency-related optimisation is to modify the selection of locations in plans 1c and 3 to select a[5] *closest* location, rather than an arbitrary (random) location. We term this improvement "opt2".

This simple robot was implemented [6], including the two optimisations (opt1 and opt2) just discussed. In addition to implementing the actions and the robot's program, the implementation included:

- Error checking of a range of conditions (such as the robot going into an infinite loop, the pre-conditions of actions not being met, etc.) that should not occur.
- Tracking the (Manhattan[7]) distance that a robot travelled (in order to allow for the effects of the two optimisations to be measured).
- Random initialisation of the starting configuration.

Note that although the program in Figure 12.1 is given in an agent-oriented programming language, the implementation used for experimental purposes was actually done in a general purpose scripting language (namely Lua, `http://www.lua.org`).

Another elaboration that is possible is to add to the robot an amount f of fuel, and have the amount of fuel be decreased when the robot moves. The robot's behaviour would also need to include a way of refuelling when needed.

In our model the robot begins with knowledge of the locations of all the bins. This is not very realistic, and could be changed as follows. First, introduce a variable $b_r : \mathcal{P}(\ell)$ which is used to track the locations of bins known to the robot; initially $b_r = \emptyset$. Then plan 3 is split into two cases: if b_r is non-empty, then select $\ell \in b_r$; otherwise, if $b_r = \emptyset$, then instead select ℓ such that $v(\ell) = \perp$. The sense action also needs to be modified to update b_r (e.g. $b_r := b_r \cup (\{p\} \cap b)$).

Finally, the system only has a single agent and clearly it could be extended to have a collection of robots exploring the landscape. Perhaps the simplest way of

then it should go straight there, and if it knows that there is no rubbish anywhere then it should stop. Figure 12.1 includes these sub-cases.

[5] There may be more than one equally close location that satisfies the desired condition, e.g. being a bin, or having rubbish.

[6] Source code available on request.

[7] Where the distance between (x_1, y_1) and (x_2, y_2) is $|x_1 - x_2| + |y_1 - y_2|$.

doing this is just to have multiple robots roaming the landscapes, oblivious to each others' presence. However, effectiveness would be improved by having the robots share their knowledge of the environment by communicating in some way, either directly by messages, or indirectly through the environment.

12.4 Correctness Proof

We now briefly argue that the robot program given in Figure 12.1 meets the specification. Note that the following proof is informal. We do not give a formal proof for a number of reasons. Firstly, it can be argued that an informal proof of this sort is more representative of common practice than formal proofs. Secondly, for an agent program written in a "real" language, there typically do not exist tools that can assist with creating or checking formal proofs. Finally, we feel that expanding this section to be formal would not add much to this chapter, but would distract from the key points in the following sections.

The proof proceeds by defining a numerical measure of progress, and then arguing that the robot's program works to decrease it, ultimately reaching 0. Our metric, $\mathcal{M}$, is defined as follows. Suppose that a given starting configuration has at most N units of rubbish in a given location, that is, $\forall \ell : v(\ell) \leq N$. For each location ℓ we let $w(\ell) = v(\ell)$ if $v(\ell) \in \mathbb{N}$, else, if $v(\ell) = \perp$, we let $w(\ell) = N+1$. Then $\mathcal{M}$ is just the sum over all locations of $w(\ell)$, with an additional term accounting for the rubbish that the robot is holding:

$$\mathcal{M} = \frac{h}{2} + \sum_{\ell \in \mathbb{L}} w(\ell)$$

Observe that $\mathcal{M} = 0$ exactly when the robot's goal, $\mathcal{G} \equiv h = 0 \wedge \forall \ell . v(\ell) = 0$, is satisfied.

We now prove that the robot's program works. Firstly, we argue that the condition on each action implies the action's pre-conditions. This is trivial: move and sense both have a true pre-condition. For pick we require that $h < c$, which follows from $h = 0$ (the condition of plan 2) and $c > 0$, and for drop we require that $p \in b$ which follows trivially from the condition of plan 4.

Secondly, we argue that the effects of the robot's actions are to monotonically decrease the metric $\mathcal{M}$. Certain actions have the effect of directly reducing $\mathcal{M}$. For pick we observe that $v(p)$ is reduced and h increased by the same amount, since $\mathcal{M}$ counts $h/2$ this yields a reduction. For drop we observe that the only change is a reduction in h. For sense we note that the action is only performed where $v(p) = \perp$, and this thus reduces the value of $w(p)$ from $N+1$ to N or less.

The remaining action, move, does not affect $\mathcal{M}$, but whenever the robot moves, it creates a situation where the following action is not a move. Specifically, we have the following three cases:

1. If the robot moves as a result of plan 1c then it moves to a location such that $v(\ell) > 0$ and the resulting state has $h = 0$ (unchanged from the condition of plan 1c) and $v(p) > 0$ and hence the next plan to apply is plan 2 and a pick action will be done.
2. If the robot moves as a result of plan $1c'$ then it moves to a location such that $v(\ell) = \bot$ and the resulting state has $v(p) = \bot$, and so a sense action will be done (plan 1a).
3. Finally, if the robot moves as a result of plan 3, then the resulting state has $h > 0$ (unchanged from the condition of plan 3) and $p \in b$, and thus the next action to be performed is a drop (plan 4), or, if the robot doesn't know how much rubbish is at the bin's location, a sense followed by a drop.

Since a move does not increase $\mathcal{M}$, the robot cannot perform a sequence of movements, and all other actions do reduce $\mathcal{M}$, we have that the robot's actions progressively reduce $\mathcal{M}$. This allows us to conclude that the robot will eventually reach $\mathcal{M} = 0$ at which point its goal is achieved.

12.5 Issues

Suppose that we use model checking or a formal proof to show that the robot program ("$\mathcal{P}$") presented in the previous section meets the system's specification ("$\mathcal{S}$"). In this section we consider a range of issues associated with doing so.

Firstly, we consider issues to do with capturing the right specification $\mathcal{S}$. It turns out that in practice the notion of correctness isn't always that easy to capture formally, even for such a simple case study. We discuss this issue below in section 12.5.1.

However, even if we do manage to capture the right specification, an issue in verifying that program $\mathcal{P}$ meets specification $\mathcal{S}$ is that it only considers the program and a specification: it doesn't consider the broader context and such factors as human errors and hardware errors. Additionally, proofs tend to be abstract, and it turns out to be easy to have hidden implicit assumptions in models or proofs, which can be dangerous. We discuss this issue in section 12.5.2.

It is worth noting that whilst some of the second group of issues are generic, i.e. they apply to any software system, not just agent systems, the nature of the "broader context" is different for agent systems. Furthermore, the first group of issues is agent specific. For instance, since agent systems are often situated in failure-prone environments where success cannot be guaranteed, the form of the specification needs to change from requiring success, to only requiring success if success is actually possible.

12.5.1 Problems with Specifications

Typically specifications are expressed in temporal logic (or an extension thereof), and there are a range of standard properties that are specified and checked against (e.g. liveness, lack of deadlock). Indeed, there are libraries of commonly used specification patterns [153].

However, capturing the right notion of "correctness" is not always straightforward with agent systems, even for the very simple case study that we consider. For instance, one common pattern, which corresponds to a goal of achieving a desired property ϕ, is to require that ϕ eventually holds ($\Diamond\phi$). However, for agent systems which may be deployed in a failure-prone environment, there may be situations where failure cannot be avoided, and so $\Diamond\phi$ is too strong, and will not be provable. For example, part of the environment may be unreachable, due to blocked paths, or due to the robot not being able to hold enough fuel. A more appropriate specification is that the robot succeeds "where possible". However, specifying the "where possible" condition is not easy. Furthermore, it depends on the agent program: an agent that is able to plan ahead and realise that it needs to refuel before heading out to retrieve some rubbish will have different failure conditions to a robot that just heads out and refuels when it is close to running out [152].

In order to capture a suitable correctness condition we turn to *dynamic logic* [347], in which *action expressions* may be primitive actions a, sequences of actions $a_1;a_2$, zero or more iterations of an action expression a^*, or a choice of action expressions $a_1 + a_2$. Then $\langle A\rangle\phi$ is read as "after performing action expression A the property ϕ *may* hold"; and $[A]\phi$ is read as "after performing action expression A the property ϕ *must* hold". Now suppose that the desired goal is $\mathcal{G}$, that the actions available to the agent are $A = a_1,\ldots,a_n$, and that the set of all action sequences is A^*. We denote the robot's program (or, more precisely, its translation into dynamic logic) by $\mathcal{P}$, and any assumptions that are being made about the initial state are denoted by $\mathcal{I}$.

In order to capture the requirement that the robot must succeed we can write $\mathcal{I} \Rightarrow [\mathcal{P}]\mathcal{G}$, but, as discussed earlier, this is too strong. What we want to capture is that the robot is only required to succeed if, given the initial assumptions, success is actually possible. The notion that "success is possible" is formalised as the existence of a sequence of actions that realises the goal, thus the overall requirement is written as:

$$(\mathcal{I} \wedge \exists A \in A^*.[A]\mathcal{G}) \Rightarrow [\mathcal{P}]\mathcal{G}$$

This formalisation only requires the agent to succeed if success is possible, which is what we want. However, the definition of "success is possible" ($\exists A \in A^*.[A]\mathcal{G}$) is not quite right. In certain situations success may be theoretically possible but not practically possible. These situations occur when there exists a sequence of actions that succeeds (which meets the definition of "success is possible"), but where the information available does not allow one to select which actions to perform. For example, suppose there are two doors, exactly one of which hides a large prize, and we have one chance to open a door and claim whatever lies behind it. Then there exists an action that claims the large prize, and hence the definition of "success is possible" is met. However, given the information available, we are not able to reliably select the correct action. We believe that this issue can be fixed by specifying a correct formalisation of "success is possible", but this involves introducing a representation for the information available, and is a diversion from the main point of this chapter.

Another issue with a specification of the form $\Diamond\phi$ is that "eventually" can be soon, or in a very long time, and in a real deployed system, knowing that the system will "eventually" achieve some desired state isn't enough: we need to know that "eventually" will arrive "reasonably soon". This issue is (somewhat) specific to agent systems. Consider a classical concurrent system, such as a printer spooler. In such systems efficiency is not typically an issue: if there are no deadlocks, then the system will complete spooling print jobs in a timely manner. However, an agent system such as a manufacturing scheduling system, or indeed, a robotic cleaner, may take too long to run, even if there are no deadlocks.

Unfortunately, "reasonably soon" can be relative to the problem instance: how long a cleaning robot could reasonable be expected to take depends on the amount and distribution of rubbish in the environment. Thus, defining the desired property of "reasonably soon ϕ" requires specifying how long an ideal robot would take to clean a given environment[8].

Although efficiency and performance issues are often dealt with through means other than formal verification, in some systems performance is critical. In these systems we often do need to provide strong guarantees about performance, and its variability, in a way that cannot be met by testing with sample cases. We return to this issue, in the context of the case study, in section 12.8.2.

In summary, getting the specification right for an agent system, so that it captures both what is actually needed (e.g. "reasonably soon" rather than "eventually") and also isn't too demanding ("succeeds where possible" rather than "always succeeds") is not easy, and specifications that are short and simple (e.g. $\Diamond\forall\ell.m(\ell) = 0$) become more complex when these issues are taken into account. However, if these issues

[8] To specify "reasonably soon" we first define the following notion of cost: given a sequence of actions A, its cost is denoted by $cost(A)$ (where the function $cost$ maps an action sequence to a natural number). This notion can be generalised to an action expression A_E (or program $\mathcal{P}$) by defining the cost of an action expression A_E executed in starting state S_0 as being the cost of the sequence of actions that is performed, that is, $cost(\mathcal{P})$ when the program is executed in starting state S_0 is defined as $cost(A)$ where A is the sequence of actions that the program performs when executed in S_0. We then define the most efficient action sequence $S_0 \in A^*$ as being a solution for the goal $\mathcal{G}$, with the additional condition that any other solution, A, has a higher (or equal) cost. We formalise this by firstly defining a solution $S \in A^*$ of a goal $\mathcal{G}$: $solution(S,\mathcal{G}) \iff (\mathcal{I} \wedge \exists A \in A^*.[A]\mathcal{G}) \Rightarrow [S]\mathcal{G}$ and then specifying the best solution S_0 as: $best(S_0,\mathcal{G}) \iff solution(S_0,\mathcal{G}) \wedge (\forall S \in A^*.solution(S,\mathcal{G}) \Rightarrow cost(S) \geq cost(S_0))$ We can now define a "reasonably good" solution as being one that is within some desired factor N of the best possible solution (clearly this is just one possible notion of "reasonably good"): $good(\mathcal{P},\mathcal{G},N) \iff \forall S_0 \in A^*.best(S_0,\mathcal{G}) \Rightarrow cost(S_0) * N \geq cost(\mathcal{P})$

are not taken into account, then a proof may not be possible (because the robot in fact cannot deal with all configurations of the environment) or may establish a result that looks nice ("it always eventually ...") but in fact is too weak to be practically useful.

Note that this discussion has focussed on the *form* or *structure* of the specification, e.g. using $\Diamond\phi$ as opposed to $(\mathcal{I} \wedge \exists A \in A^*.[A]\mathcal{G}) \Rightarrow [\mathcal{P}]\mathcal{G}$. A related, and well-known, issue is getting the *contents* of the specification right (i.e. the choice of *which* ϕ) [261].

It is also worth noting that the point of this subsection is not to argue that the correctness specification should be exactly as described, but to highlight that attempting to address the two key issues of using simple and natural specifications such as $\Diamond\phi$ results in a significantly more complex specification. Furthermore, and perhaps more importantly, the resulting more complex specification appears to be rather more challenging to verify (but we have not verified this yet).

12.5.2 Problems with Proofs

Beware of bugs in the above code; I have only proved it correct, not tried it — D. Knuth[9]

As noted earlier, the enterprise of verification is concerned with showing that a program $\mathcal{P}$ meets its specification $\mathcal{S}$. The previous subsection discussed a number of issues with getting the specification $\mathcal{S}$ right. However, even if we can get the specification right, there are still two significant issues with proving that a program meets its specification:

1. Showing that a program meets its specification does not consider the wider context: despite a correct proof being given, issues may still arise due to human interaction, or physical interference. This motivates the argument (in section 12.7) that we need to *broaden* the scope of verification.
2. A proof may be abstract and may contain implicit (typically "obvious") assumptions that turn out to fail, making the proof wrong. Knuth's comment highlights that, in some cases, these "obvious" assumptions can be detected very easily by testing, which motivates the argument (in section 12.8) that we should look at ways to combine testing and proving techniques.

The first issue is that traditional approaches to verification consider only the program, ignoring other issues, such as user interfaces, human errors, and hardware faults. The assumption that seems to justify the narrow focus on programs and (formal) specifications seems to be that software errors — that is, differences between the (formal) specification and the implementation — are the key issue.

[9] http://www-cs-faculty.stanford.edu/~knuth/faq.html

However, although software errors clearly are significant, analysis of computer-related failures suggests that software errors are only a part, and possibly a rather small part, of the problem. For instance, an analysis[10] of computer-related deaths[11] [294, Chapter 9] found that, of roughly 1,100 deaths[12] that were caused by computer failures up to 1992[13] [295, Chapter 9, p300]:

> ... Over 90 percent of these deaths were caused by faulty human-computer interaction (often the result of poorly designed interfaces or of organizational failings as much as of mistakes by individuals). Physical faults such as electromagnetic interference were implicated in a further 4 percent of deaths, while the focus of Hoare's and Licklidder's warnings, software "bugs", caused no more than 3 percent, or thirty deaths: two from a radiation-therapy machine whose software control system contained design faults, and twenty-eight from faulty software in the Patriot antimissile system that caused a failed interception in the 1991 Gulf War.

Writing about these results, Jackson [250, Pages 86-87] notes that (emphasis added):

> ... coding errors were cited as causes only 3% of the time. **Problems with requirements and usability dwarf the problems of bugs in code, suggesting that the emphasis on coding** practices and tools, both in academia and industry, **may be mistaken**.

That is, the vast majority of the time (97%), computer-related deaths were not due to problems in coding, and would not have been caught by formal verification of implementation against specification. The key point here is that in order to cover more than 3% of MacKenzie's cases, the scope of the specification needs to be broadened to include social, organisational, and human aspects of the system.

We have already seen (in the previous section) that getting the specification to reflect the real need is particularly challenging for agent systems. When verifying agent systems, which are situated in an environment, it is important to capture this environment. Furthermore, for agent systems that support human activity, such as disaster response coordination [389], or space exploration [69], it is important to capture the human and organisational context of the system as part of a specification. In the next section we will discuss some examples of verification that considers human aspects. However, the bulk of the work on agent verification does not consider humans to be within the scope of the specification.

The second issue with proof is assumptions: all too often assumptions made are hidden, rather than made explicit. An excellent example, in a very simple setting, is binary search [250, Page 87] (footnotes and emphasis added):

---

[10] The original paper was published in *Science and Public Policy* in 1994, and was re-printed as Chapter 9 of [294]. A less detailed discussion of the results of the analysis appeared in Chapter 9 of [295].

[11] More precisely, it focussed on "computer-related accidental death", where "computer-related" indicates a careful analysis of whether the presence of computers was a causal factor in the death(s), and where "accidental" excludes deaths caused by military computer systems that are designed to kill (the analysis also, reluctantly, excludes any associated "collateral" civilian deaths).

[12] MacKenzie's analysis focussed on deaths, because deaths are clearly defined (unlike injury, which would include RSI), and because deaths are more likely to be reported.

[13] The analysis only considered data up to 1992; however, there does not appear to be any more recent analysis.

> Proof is not foolproof, however. When a bug was reported in his own code (part of the Sun Java library), Joshua Bloch found[14] that the binary search algorithm—proved correct many years before (by, amongst others Jon Bentley in his *Communications* column) and upon which a generation of programmers had relied—harbored a subtle flaw. The problem arose when the sum of the low and high bounds exceeded the largest representable integer[15]. Of course, the proof wasn't wrong in a technical sense; there was an **assumption** that no integer overflow would occur (which was reasonable when Bentley wrote his column, given that computer memories back then were not large enough to hold such a large array). In practice, however, **such assumptions will always pose a risk, as they are often hidden in the very tools we use to reason about systems** and we may not be aware of them until they are exposed.

In the next section we consider this issue in the context of our waste disposal robot case study and explore where implicit assumptions have been made.

12.6 Assumptions in the Waste Disposal Robot Case Study Revisited

Considering the earlier proof that the robot program meets its specification (section 12.4), it turns out, in fact, that the proof isn't quite right. It makes a number of assumptions without stating them. These assumptions may seem reasonable, but they may be false. Worse, because the proof is abstract it is easy to miss these assumptions (did *you* spot all of them?).

The first (implicit) assumption is that there are bins (i.e. that $b \neq \emptyset$). If this assumption is violated, then, unless there is no rubbish in the environment, the agent will fail: once it has located and picked up rubbish, it will attempt to use plan 3, and there is no way to select an ℓ such that $\ell \in b$ if $b = \emptyset$.

A related issue is the other places where the robot selects a location ℓ that satisfies some condition (in plans 1c and 1c'). Do we have a similar issue there? The precondition for plan 1c includes the condition $\neg\forall\ell.v(\ell) = 0$ which can be rewritten as $\exists\ell.v(\ell) \neq 0$. Thus, the precondition guarantees that there will exist at least one location ℓ such that $\ell > 0$ or $\ell = \perp$, so the "select ℓ such that ..." cannot fail. But actually this reasoning also relies on an assumption: that the value of $v(\ell)$ will be either $\perp$ or a natural number. If this assumption doesn't hold, then it is in fact possible for there to not be any possible selection for ℓ that satisfies the conditions of plan 1c (and this in fact did occur in testing — see below).

The second key assumption is that the robot's view, v, "mirrors" the real environment, m. More precisely, that $\forall\ell.v(\ell) = m(\ell) \vee v(\ell) = \perp$, that is, for each location, the robot's view is either unknown ($\perp$), or matches the environment. In our experimentation we considered a range of initial situations where this assumption was

[14] http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

[15] That is, code of the form `middle = (high + low) / 2`.

violated. However, an inconsistency between v and m can also arise *during* execution because of errors in sensing or acting (thinking you've picked up rubbish when in fact you haven't), and also because of exogenous activity (such as other robots cleaning rubbish, or pesky humans littering their environment).

We now explore a range of issues that can arise when, for some location ℓ, $m(\ell) \neq v(\ell)$ (and $v(\ell)$ is not $\perp$). It is worth emphasising that **all of these issues have been observed in the implementation.**

Case 1: One case is where $m(\ell) > 0$ but $v(\ell) = 0$. In this case the robot incorrectly believes that location ℓ is clean, and the robot will, in fact, never visit the location (since there is no need), and, if there are no other issues, will complete execution believing that it has successfully cleaned the world, whereas there is still rubbish at ℓ.

Case 2: Another case is where $m(\ell) = 0$ but $v(\ell) > 0$, i.e. the robot believes there is rubbish at ℓ, but in fact there isn't. In this case the robot will eventually (if no other errors intervene) arrive at ℓ and proceed to pick up the rubbish. The pick action changes $m(\ell)$ and $v(\ell)$ by t, which in this case is zero, i.e. the pick action has no effect. This means that the situation is not changed, and the next plan that is applicable remains plan 3, and the robot will again attempt to pick up the non-existent rubbish, leading to an infinite loop.

Note that, in fact, any situation where $v(\ell) > m(\ell)$ will end up with a loop, since (if no other errors intervene), the robot will pick rubbish, eventually reducing $m(\ell)$ to zero.

Case 3: We now turn to the situation where both $m(\ell)$ and $v(\ell)$ are greater than zero, but where $m(\ell) > v(\ell)$. A range of behaviours can result from this situation. We begin by observing that if $m(\ell)$ and $v(\ell)$ are both greater than 0, then pick action(s) will reduce both of them until eventually $v(\ell)$ is not greater than zero, resulting in one of the following sub-cases:

- If $v(\ell) = 0$ then we are left with the first scenario discussed above: the robot will incorrectly believe the location to be cleaned, and (if no other errors intervene), will eventually believe its goal to be completed, even though rubbish remains in the environment.
- If $v(\ell) < 0$ then we have an invalid value, i.e. the assumption that $v(\ell)$ is either a natural number or $\perp$ does not hold. The implementation actually uses -1 to represent $\perp$, which results in the following two sub-cases:
  - If $v(\ell) = -1$ then in fact the robot recovers: it interprets the -1 as being $\perp$ and performs a sense action which makes $v(\ell) = m(\ell)$, resolving the difference.
  - If $v(\ell) < -1$ (e.g. $v(\ell) = -2$) then plans 1a, 1b, 1c and 2 are not applicable. This leaves plans 3 and 4. If the location happens to be a bin and the robot is holding rubbish[16] then it will drop the rubbish, at which point *none* of

[16] Which will be the case, because a pick action is what reduced $v(\ell)$ to -2.

the plans are applicable and the robot halts with an error. If the location is not a bin (and the robot is holding rubbish) then it will move to a bin and drop the rubbish there. Once the rest of the grid is cleaned, the robot is left in a situation where the only applicable plan is 1c (since it is not done), but where the "select an ℓ such that ..." cannot be satisfied: there are no remaining locations for which $v(\ell) > 0$ or $v(\ell) = \perp$: the location that causes $\forall \ell.v(\ell) = 0$ to be false has a value of -2, which is neither of these cases.

To briefly summarise these cases we have the following possible behaviours:

1. Finishing, believing the task to be completed, even though there is rubbish in the world (case 1, where $v(\ell) = 0$ and $m(\ell) > 0$)
2. An infinite loop (case 2, where $v(\ell) > 0$ and $m(\ell) = 0$)
3. Recovering because -1 is used in the implementation to represent $\perp$ (case 3 for $v(\ell) = -1$)
4. Aborting because either no plan is applicable, or plan 1c is applicable but no suitable ℓ can be selected (case 3 for $v(\ell) < -1$)
5. Being unable to execute plan 3 if no bins exist.

In our case study there are a number of further assumptions that are implicitly made. These include: that paths between locations are never blocked, that robots never break down, that rubbish is measured in discrete units that can be picked up by a single robot, that the environment doesn't change, and that the robot's navigation systems are perfect. Additionally, we assume that the environment is a grid, which clearly isn't true for a real world. If the robot operates in a real environment, then this last assumption amounts to assuming that a symbolic representation of the environment is used, and that this representation is able to be accurately determined from sensors.

The point here is not to argue whether any one of these further assumptions is reasonable or not, or whether a given assumption would have been made. The key point is that assumptions are being made, and that they are often made implicitly and not documented questioned or justified.

To summarise, we have argued that correctly capturing the specification is not trivial, and that it is easy to either require too much (by requiring the agent to always succeed), or too little (for instance requiring something to happen "eventually" without considering reasonable time bounds). Furthermore, even if the specification is correct, there are a range of factors that are not typically considered in verification (such as human interaction), and it is possible for proofs to contain implicit assumptions that render them incorrect. In the next two sections we consider how to address these issues.

12.7 A New Approach to Assurance of Agent Systems

So, what *should* we be doing to obtain assurance of agent systems? Before we discuss our approach for obtaining assurance we need to emphasize that verification has a key role to play, since, in general testing on its own is not sufficient. Although in some applications testing may be enough — for instance, recall that for one application "*Even though this negotiation process has not been theoretically proved for cycles' avoidance* [sic], *practical experiments have validated its operation*" [349, Page 413] — we do not believe that in general it is sufficient.

The reason for not believing that testing is sufficient is twofold. Firstly, it is known, and widely accepted, that concurrent systems, which are able to exhibit time-dependent errors, are challenging to test. Multi-agent systems are concurrent systems. Worse, they are concurrent systems where the individual components (the agents) are able to behave flexibly and adaptively. This makes agent systems *harder* to test than other concurrent systems. Indeed, an analysis of the state space size for BDI systems [436] found that the space of possible behaviours was extremely large (e.g. in excess of 10^{100} for a uniform depth 3 tree, and over 10^{11} for a sample tree from an industrial workflow application). Tsai *et al.* report on similar analyses for knowledge-based systems, and conclude that [419, p205–206]:

> ... for systems that use either a selection method (such as MYCIN and INTERNIST) or the construction method (such as XCON, XSEL, and XFL), the potential sizes of the input and the output spaces for black-box testing are enormous.

Secondly, testing is not compositional, whereas proof can be. If we have a goal G which is decomposed into two sub-goals G_1 (achieved first) and G_2 (achieved second), then if we have tested G_1 and G_2 separately, it is not clear what we can conclude about G: the situations that result from achieving G_1 may not relate to the situations in which G_2 was tested. On the other hand, if we prove that G_1 always succeeds, and the assumptions that are needed to prove that G_2 succeeds are a consequence of the success of G_1, then separate proofs of the correctness of G_1 and G_2 *can* be combined to provide a proof of the correctness of G.

The remainder of this section briefly describes a proposed solution to obtaining assurance regarding the behaviour of a multi-agent system. As might be expected, the proposed solution is sketched briefly, and is somewhat tentative: more work is required.

The solution has two aspects: adopting a more pragmatic approach that assesses risks, and uses appropriate levels of mitigation and evidence of correctness (section 12.7.1); and combining testing and proving [17] (section 12.8). We also briefly discuss the issue of programming languages and approaches (section 12.7.2).

[17] In the remainder of this section we use the term "proving" broadly, to encompass any approach that (unlike testing) considers *all* possibilities; specifically, we use "proving" to encompass both mathematical proofs and model checking.

12.7.1 An Engineering Approach to Risk Management

Based on the issues highlighted, we feel that we should look at a more broad engineering solution: we need to adopt an engineering approach that quantifies the risk and then uses appropriate levels of evidence[18], as is argued by Jackson [250, Page 81]:

> ... as in all engineering enterprises, dependability is a trade-off between benefits and risks, with the level of assurance (and the quality and cost of the evidence) being chosen to match the risk at hand.

Jackson argues for the use of *direct* evidence that a system meets its requirements: an end-to-end argument that provides evidence that the system exhibits desired properties. Jackson also argues that properties should be expressed in real world terms rather than in software terms. For example, specifying a safety property in terms of the radiation dose received by a patient, rather than in terms of the software computing a correct dose. This tends to encourage consideration of the wider context, for instance, how a radiation dose computed by the software is translated into a radiation dose that is delivered to a patient.

Rushby [378] also argues for a safety case that formally establishes that a claim follows from the system and (explicit) assumptions. He gives an example of an adaptive system, and proposes that verification before deployment be augmented with run-time monitoring of assumptions, which may be formally derived. It is worth noting that although he discusses "adaptive systems", these are not agent systems, and it is not clear whether agent systems are as complex, simpler, or more complex than the systems he discusses.

It is noteworthy that increasingly, the use of safety cases is becoming accepted, as reflected in government standards. For example, Bishop *et al.* [46, section 4.2] discuss a range of UK standards that have adopted safety cases, including the use of goal-based safety cases. The goal-based approach proposed by Bishop *et al.* [46] derives desired safety goals using a range of techniques, such as hazard analysis, and then provides *evidence* that supports the desired *claims* via *arguments*. A range of forms of evidence are used, including "*deterministic or analytical application of predetermined rules*" which covers formal proofs, as well as probabilistic analyses, and process-based arguments. Goal-based assurance is supported by a range of tools and notations, including the Goal Structuring Notation (GSN) [270]. Note that this work is not specific to software, let alone to agent software. It is clear (from earlier discussion in this paper) that assuring agent software presents particular challenges that affect the choice of claims, evidence and arguments; however, safety cases and goal-based assurance can still form a useful general framework for the assurance of agent systems.

[18] As a aside, it is interesting to observe that much of verification research is "European", whereas a more pragmatic approach based on risk management is more "American". One might speculate on whether this difference is cultural, or a result of the funding landscape, or of other factors, such as the relationships between academia and industry [147].

More broadly, there are a range of techniques for assessing possible failure modes and their risks (e.g. fault trees, event trees), but these have barely been used with MAS (an exception is [136]). These methodologies can allow assumptions to be identified, and can allow us to develop an *error model* which captures the errors that need to be dealt with.

In the context of our simple case study, applying one of these processes, perhaps a goal-based assurance case approach [46], would lead us to think about the wider context, and to realise that the desired top-level goal is not that the robot sees the world as being clean ($v(\ell) = 0$) but rather, that the world itself is empty of trash, i.e. $\forall \ell . m(\ell) = 0$.

One process for developing (formal) requirements that considers the larger context is that of Jones *et al.* [262]. They propose a design methodology for deriving the specification of the software-to-be from a specification of its environment. They present a design methodology that starts by capturing formally the “problem world”, that is, an abstraction of the world in which the software is situated. The interface between the software-to-be and the problem world is specified abstractly in terms of interfaces, and “rely conditions”: what conditions the software can rely upon; for example, that sensors and affectors work correctly. Doing this captures assumptions explicitly. A feature of the approach is that fault-free operation and faulty operation are dealt with separately.

In the context of the case study, an attempt to document the interface between the robot and its environment and explore the “rely conditions” would lead us to ask what grounds we have for believing that the robot’s view matches reality, thus uncovering an (implicit) assumption.

We have argued that there is a need to broaden the scope of verification to consider humans. We now consider some examples of how verification techniques can be applied to verify systems including both software and humans.

The work of Bordini *et al.* [69] considers systems that involve collaboration between humans and software agents, specifically space missions. Agent teamwork is specified in *Brahms*, a language that has been developed over a number of years for modelling human activity. The key issues in verifying human-agent teamwork specified in Brahms are that the Brahms notation is quite complex, and that it lacks formal semantics. Three possible approaches are briefly discussed. One is to simply use Java Pathfinder (JPF) and run the Brahms implementation on top of this, but there are efficiency issues with doing this. Another approach is to reimplement Brahms within the AIL framework [140].

A third approach, which has some resemblance to the approach that is proposed in this chapter, is to use a stepping stone approach: translate Brahms models into Jason (using abstraction, so the Jason model isn’t a precise re-implementation of the Brahms model). Then the Jason implementation can be formally verified, and runs of the Brahms model can be compared with runs of the Jason implementation.

The work of Rushby *et al.* [118, 376, 377] uses formal methods techniques to look for mode errors in cockpit interfaces, i.e. situations where a human pilot (who may be operating under stress) is likely to mistakenly believe the system to be in a particular state when it is actually in a different state. This was done by modeling the system (e.g. as a finite state machine), modeling the human's mental model of the system (e.g. elicited from human pilots), and then using model checking to find divergences.

More broadly, there is work (e.g. [119,151]) that uses formal methods techniques to model human behaviour, and then reason about it. As Duce *et al.* [151, Section 3.3] note:

> ... safety critical systems typically involve human agents as well as computer agents, and once again we see that to be able to reason about the overall properties of the system we need to be able to reason at some level about the human agents in them.

This approach has begun to be explored in non-agent safety critical systems, but does not seem to have been considered in the context of agent systems.

12.7.2 "Send considered harmful?"

Finally, perhaps a more minor, but nonetheless important point, concerns the level of programming. We should aim to work at a level that avoid certain error classes. In the same way that avoiding manual memory allocation in favour of garbage collection avoids a whole class of errors, we should seek to work at a level that allow us to avoid error classes. For agents one particularly place to consider this issue concerns concurrency: agent systems are concurrent, but some concurrency-related errors could be avoided by working at a higher level than message sending and receiving. Building multi-agent interactions in terms of sending and receiving individual messages is error-prone, and could be argued to be analogous to programming based on "goto" statements. A similar argument has been independently put forward by Chopra and Singh [96].

A number of alternative ways of specifying and implementing interactions have been proposed in the agent literature (e.g. [95,97,273,434,435,443]). What these approaches all have in common is that although they ultimately do realise communication by sending messages, the interaction is specified and implemented in terms of higher level constructs, and certain errors simply cannot occur as a result of this. For example, interactions that are designed (and implemented) in terms of social commitments are able to ensure "alignment" in the face of concurrency. Although different agents may perceive a different ordering of messages, under certain conditions, they will reach the same conclusions about the commitments that hold [97,434,435].

12.8 Combining Testing and Proving

As we have argued earlier, neither testing nor proving are sufficient on their own. However, each have strengths, and they can be used to complement each other. This is evidenced by the results from our case study: testing uncovered assumptions that the formal proof missed, but the formal proof covers all cases, which testing cannot. A similar argument about the complementability of testing (in the form of random search [329]) and model checking was presented by Gao *et al.* [187] who experimented with a specification that had been randomly injected with faults, and found that their Lurch tool found some bugs that the SPIN model checker didn't find (they also found that most of the randomly injected faults were easy to find using random testing).

So, we should look at combining testing and proving in a way that allows them to work effectively together. But how can we use testing and proving together in a meaningful way?

In this section we outline an approach for combining testing and proving. The proposed approach is generic, in that it applies to a range of software, not just agent systems. We believe that generality is an advantage, and that what is important is not whether the approach is *specific* to agent systems, but whether it is *applicable* to agent systems. It is possible for a generic approach to fail to be applicable, or to fail to be useful, in a more specific context.

The proposed approach is based on the recognition that testing and proving are merely two possible ways of trying to provide evidence of correctness, and that there are other, intermediate, approaches. We then use these intermediate approaches to build a "bridge" between testing and proving.

Testing and proving are just two particular techniques amongst many, which we classify along two dimensions (see Figure 12.2, ignore the arrows for now). The first dimension is *abstraction*: does the technique deal with the actual code that is running ("concrete"), or with an abstract model ("abstract")[19]? Specifically, an "abstract" model is one where the actual running code is not derived in an automated way from the model. We distinguish between "concrete" and "abstract" because an analysis of a concrete model tells us something about the actual running code, whereas analysis of an abstract model is one step removed from the actual executing code. The second dimension is the *coverage* of the state space: does the checking cover only certain points (testing)? does it cover all points within a sub-space (incomplete systematic exploration)? or does it cover all of the state space (complete systematic exploration)? We do note that some of these lines are somewhat imprecise: for instance, Java PathFinder does systematic exploration of Java code ("concrete"), but some approximations need to be made, so it's not completely concrete.

Testing and proving are familiar, but some of the other approaches in Figure 12.2 need to be briefly explained:

[19] We also have a "very abstract" classification for dynamical systems techniques.

Shallow Scope: The idea is that all possibilities are explored systematically (i.e. like model checking, rather than like testing) but only within a limited scope for variable values. For instance, we might consider $\mathbb{L} = \{0\ldots2\} \times \{0\ldots2\}$ and a maximum value for $m(\ell)$ or $v(\ell)$ of 1. The promise of this approach is the "shallow scope hypothesis" which states that many errors in models can be found with a fairly small scope. Experimentation with a range of models has provided evidence for the shallow scope hypothesis (e.g. [138, 249])

Systematic Enumeration: Systematically generating all possibilities, within a given scope. The difference between this and "Shallow Scope" is that it is done with an implementation, which may make it harder to work symbolically, or to map to other representations such as Ordered Binary Decision Diagrams. For example, in our case study, we could generate all possible starting configurations for a limited-size grid; and then execute them with the implementation.

Animation: Roughly speaking, executing specifications [269, 363]. Unlike executing programs, this may be possible only for some specifications (because not all specifications are executable), and may involve analysis to try and execute ("animate") specifications that are not in a convenient form to be executed directly.

Dynamical Systems: This approach is very abstract and considers the overall behaviour of the space of possible executions, viewed as a dynamical system. A typical question is whether there are attractors, and what is their basin of attraction. However, although this approach has promise, and is worthy of further work, there seems to have been little work on using such approaches for verification in the computing community (an exception is the work of Beer [29]). One challenge is that software systems are typically discrete, whereas dynamical systems are normally continuous.

So, how do we use these techniques to build a bridge between testing and proving? The key idea is to use small steps that only change one aspect of the taxonomy of Figure 12.2. For example (the numbers correspond to the numbered arrows in Figure 12.2):

❶ Comparing the outcome of testing the implementation (with selected test cases) with the outcome of systematic enumeration (still with the implementation) allows us to assess to what extent the selected test cases are sufficient.

If both approaches find the same errors, then this gives us confidence that the test cases are in fact sufficient. If we find errors with systematic generation that we don't find with the test cases, then the test cases are inadequate. If we find errors in the selected testing that are not found by systematic generation, then this tells us that the scope of generation is too limited (this is also assessed by comparing proving and systematic generation with the abstract model, discussed below).

❷ Comparing the outcome of systematic generation with the implementation and with an abstract model allows us to assess the difference that is made by changing to an abstract model.

Coverage: / **Abstraction:**	*Individual Test Cases*	*Systematic Exploration (incomplete)*	*Systematic Exploration (complete)*
Very abstract	-	-	Dynamical Systems
Abstract Model	Animation	Shallow Scope $\xrightarrow{❸}$ ❷ ↑	Model Checking, Proof
Concrete	Testing $\xrightarrow{❶}$	Systematic Enumeration	-

Fig. 12.2 Approaches to Assurance: A two-dimensional taxonomy

If both approaches find the same errors, this gives us confidence that the abstract model corresponds to the implementation. If errors are found in the implementation but not in the abstract model (or vice versa), then this tells us that the abstract model is too abstract (or that it, or the implementation, is wrong).

❸ Comparing systematic generation within a limited scope ("Shallow Scope") with proving correctness (for the same model) gives us information on whether the scope is too limited, and hence is missing issues. If both approaches find the same errors, this gives us confidence that the scope limitation is not missing anything.

Taken together, these steps build a bridge that links proving and testing. At each step along the way we change only one thing which allows for conclusions to be drawn from differences, or lack thereof. Section 12.8.1 illustrates how these steps are applied to the waste disposal robot case study.

One particular issue is ensuring that multiple models capture the same thing. For instance, we might have an abstract formal model in one notation, and an implementation in another notation. The key idea that allows comparison is that we perform the *same* assurance approach on both models (typically systematic, but incomplete, exploration). This ensures that any differences found are due to differences between the models, not due to a difference in the assurance procedure. For example, we might be doing shallow scope exploration with an Alloy model and model checking with a Promela model. In this case, we would need to take multiple steps to move from one to the other. For instance an intermediate step might be model checking an Alloy model. Of course, our ability to do this may be limited by the available tools. Whilst systematic incomplete exploration can not provide perfect assurance that two models correspond, it can provide stronger confidence than non-systematic exploration.

Another case where more steps may be useful is to consider a small scope when following arrow ❷, and then also doing shallow scope exploration (with the same abstract model) but with a larger scope. This can help gain further confidence that the smaller scope is not too small.

12.8.1 Applying the Proposed Approach to the Case Study

We now, for illustrative purposes, discuss the application of these steps to the case study. We begin with comparing systemic generation (within a limited scope) and selected test cases, both with the implementation (arrow ❶). The implementation was extended to systematically generate all initial states and to run the robot in each of these. We considered a grid of 2×2 (and also a smaller 2×1 grid). Initial states were generated with the following ranges of values:

- Each $v(\ell)$ was either $\perp$ (represented as -1), or was in the range $(0 \ldots 3)$
- Each $m(\ell)$ was in the range $0 \ldots 3$
- The robot's carrying capacity c ranged over $1 \ldots 4$
- The initial rubbish carried h ranged over $0 \ldots c$
- The number of bins ranged from 1 to the number of locations (i.e. $1 \ldots 4$ for the larger scope). We also separately experimented with allowing for no bins, which introduced a new error when plan 3 was unable to find a bin to move to.
- The robot could begin in any of the locations.

These generated 33,600 initial configurations for the 2-location scope, and 35,840,000 initial configurations for the 4-location scope. The robot program was run in each of the starting configurations (taking just under 44 minutes[20] for the larger scope, and less than 3 seconds for the smaller scope). The results were collected and analysed, and the following errors (described in section 12.5) were found to have occurred:

1. Completing execution without having cleaned the environment.
2. Going into an infinite loop[21] because pick had no effect.
3. Recovering (incorrectly) because -1 was interpreted as $\perp$, leading to a sense action.
4. No action being applicable (because $v(\ell) = -2$).

[20] On an idle 2.4 GHz Intel Core 2 Duo machine with 1 Gig of 667 MHz DDR2 SDRAM running Mac OS X Version 10.5.6.

[21] In fact the condition that led to this was detected and lead to an abort, rather than a loop.

5. Failure of plan 3 due to non-existence of bins in the environment.

We compared these errors with those found by random testing, which was done by randomly generating initial configurations. Each location ℓ had a 50% chance of having rubbish and each $v(\ell)$ was initialised to $\perp$ (90% chance), or to a random number (10% chance), which meant that there was less than 10% chance of an error for a given $v(\ell)$ because the randomly generated number might be the same as the corresponding $m(\ell)$. Running 100 random tests found the first two errors[22]. Considering the fourth error, it only occurs when $v(\ell) = -2$, which requires an initial setup of $m(\ell) = 3$ and $v(\ell) = 1$ and a capacity of at least 3. This combination of circumstances was not very likely to occur randomly. However, running 1,000 randomly generated test cases (taking less than half a second) did find all four error types. This comparison, between systematic generation and random testing, told us that our initial number of tests *was* too limited, and gave us confidence that the new number of tests was sufficient.

We then did a comparison between systematic generation with the implementation and with an abstract model (arrow ❷ in Figure 12.2). In our case study, since the proof is informal, we wanted to do systematic generation with a model that, like the proof, directly realised the model in Figure 12.1. We thus implemented a model in Prolog (available upon request) that directly followed that in Figure 12.1, and did systematic generation[23] with both a 2×2 and a 2×4 grid. In both grid sizes the same error cases were detected[24]:

1. Completing execution correctly, but with rubbish remaining in the environment (implementation error case 1)
2. Going into an infinite loop, due to pick not having any effect (implementation error case 2)
3. Out of domain error: attempting to store a value less than 0 into $v(\ell)$.
4. Being unable to select a bin if no bins exist (implementation error case 5)

Comparing these cases with the ones uncovered by the implementation, we can see that we find the same errors, with one exception. Because the abstract model checks whether values in $v(\ell)$ and $m(\ell)$ are in the appropriate range, it catches cases where a value less than 0 would be stored in $v(\ell)$ (by pick) and gives an error, which prevents error cases 3 and 4 (in the implementation) from occurring. This comparison tells us that the implementation is not quite faithful to the model, but in terms of the situations that cause errors, and the assumptions that are required to rule out such situations, the abstract model and the implementation do seem to be in agreement.

[22] We did not consider situations with no bins, so error 5 could not occur.

[23] For the larger scope this took just under 40 minutes (on a Dell PC with a 2.66 GHz Pentium 4, 1280 MB of RAM, running Ubuntu Linux 2.6.24-19) to generate 3,346,110 cases (with the assumptions not allowed to be violated, i.e. $b \neq \emptyset$ and initially $\forall\ell.v(\ell) = \perp$); whereas if these assumptions could be violated it took just under 1 hour and 46 minutes to generate 13,436,928 cases.

[24] If the assumptions that initially $\forall\ell.v(\ell) = \perp$ and that $b \neq \emptyset$ held then there were no errors.

Finally, considering the third link (arrow ❸ in Figure 12.2), we observe that the errors found by systematic generation in the abstract model catch the assumptions that render the proof faulty, and indeed, if we require these assumptions to hold, then systematic generation does not find any errors, which gives us additional confidence that the proof is now correct.

To summarise, we have applied the proposed "bridge building" mechanism to the case study and found that:

- 100 test cases (with the given parameters) were insufficient, but 1,000 test cases found the same errors as systematic generation within a limit scope. This gives us some confidence in the coverage of the tests.
- Systematic generation of tests cases with the implementation and the abstract model found the same error classes (with differences that highlighted the difference in error checking, and the use of -1 to represent $\perp$ in the implementation). This gives us some confidence that the abstract model and the implementation are capturing the same thing, and gives us some confidence in applying results about the abstract model to the implementation.
- Systematic generation of tests cases with the abstract model for both large and small scopes found the same error cases, which suggests that the small scope is sufficient.
- Comparing systematic generation with the abstract model and the proof of correctness given in section 12.4 we find that the systematic generation identifies the assumptions that the proof makes. Furthermore, the proof provides additional confidence that the correct behaviour observed in systematic generation with the assumptions will generalise and hold in all settings, not just within the limited scope of generation.

Taken together, these results give us confidence in the correctness of the robot program, by providing end-to-end evidence of correctness. More importantly, this process identifies the assumptions required for the proof.

12.8.2 Addressing Efficiency

As per the discussion in section 12.5.1 we do need to take care with specifications, in order to avoid having specifications that are too strong (requiring success, even though it may not always be possible), or too weak (only requiring success "eventually", rather than "reasonably soon"). For our case study the former is not relevant, because in this case study it is always possible to succeed. However, finishing within a reasonable amount of time *is* a possible issue and so we now consider efficiency.

There are a number of ways of looking at efficiency, and perhaps the simplest is to use the implementation, rather than the model. Of course, running the implementation with selected test cases is not very convincing: it may be that particular situations result in much poorer performance, and certain applications do require a stronger guarantee on performance bounds than can be obtained with testing alone. For instance, we may need to be confident that our waste disposal robot will be able to clean a grid of a certain size within a certain amount of time. So, we use systematic generation to consider all possible initial configurations within a narrow scope. We use a 2×2 grid, where each $m(\ell)$ was in the range $0 \ldots 3$, capacity c ranged over $1 \ldots 4$, initial rubbish held h over $0 \ldots c$, and the robot could begin in any of the locations. Since the question is, "if the robot completes, how long does it take?" we avoid errors (which would lead to non-completion) by enforcing the two assumptions discussed earlier. Specifically, we ensure that there are bins ($b \neq \emptyset$) and we initialise each $v(\ell)$ to $\perp$. We tracked the (Manhattan) distance travelled by the robot. This generated 57,344 initial configurations with the following efficiency:

- With no optimisations enabled ("noopt") the average distance travelled by the robot was 6.18. The minimum[25] distance was 3 and the maximum[26] was 26.
- With only the first optimisation enabled ("opt1") the average distance travelled by the robot was 6.23. The minimum was 3 and the maximum was 26.
- With only the second optimisation enabled ("opt2") the average distance was 5.35; the minimum and maximum were respectively 3 and 26.
- With both optimisations enabled ("optboth") the average distance travelled was 5.25; the minimum and maximum were 3 and 26.

Figure 12.3 shows the efficiency in terms of distance travelled in graphical form. The horizontal axis shows the distance travelled, and the vertical axis counts how many of the 57,344 cases required that given distance to be travelled. As can be seen, most of the cases don't involve a large distance, but there are a few that do. For instance, with no optimisations, around 93% of the possible initial configurations require traversing a distance of 10 or less; and around 99% of configurations require traversing a distance of 17 or less.

We also considered a 3×2 grid, for which the average distance was 14.6 without optimisations ("noopt") and 11.75 with the second optimisation ("opt2"), and the minimum distance was 5, and the maximum was 56 for both. As is shown in Figure 12.4, the distribution is similar to the 2×2 case. With no optimisations around 90% of configurations involve covering a distance of 21 or less, and 99% of configurations involve a distance of 36 or less (with the second optimisation these numbers are respectively 19 and 34). As in the 2×2 case, the maximum distance of 56 is very unlikely (4 cases out of 917,504 or 0.000436%).

Overall, from this exploration of efficiency we can conclude that, for the case study, there *are* cases where the effort required (in terms of distance) is significantly

[25] Since the robot did not begin with knowledge of the environment, even if there was no rubbish, it still had to cover the area to find this out.

[26] A distance of 26 may seem high for a 2×2 grid: it arose in situations where there was much rubbish (i.e. $m(\ell) = 3$ for at least locations 2 to 4), the robot had a low carrying capacity ($c = 1$), and only location 1 was a bin. This was a relatively rare situation (4 cases out of 57,344, or less than 0.01%).

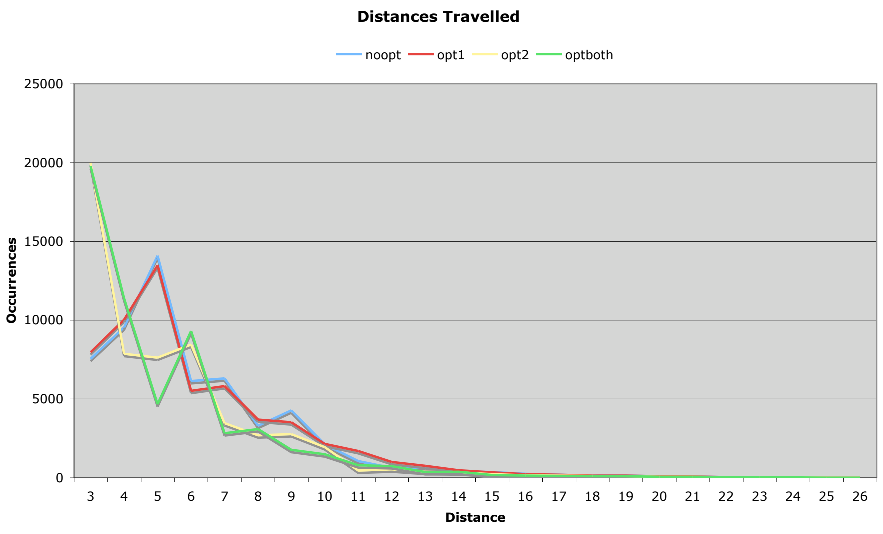

Fig. 12.3 Efficiency Profiles for 2×2 grid.

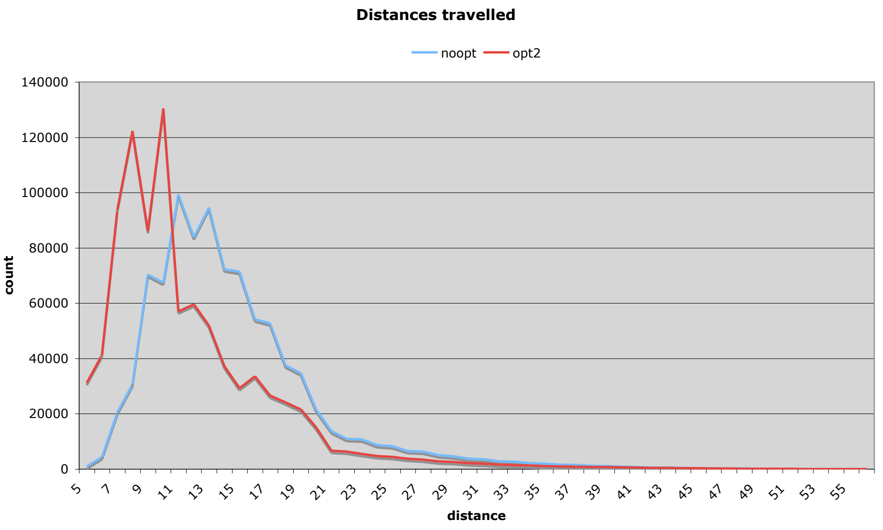

Fig. 12.4 Efficiency Profiles for 3×2 grid.

worse than the average or commonly encountered cases, and that consequently these cases are unlikely to be found through random testing. On the other hand, we have a bound on how much worse these cases are: the maximum distance travelled was around 3.8 to 4.95 times the average distance.

12.9 Conclusions

In this chapter we considered the issue of obtaining assurance that an agent system will behave appropriately when it is deployed. We asked what role formal verification are able to play in this process. We identified a number of issues relating to the difference between assurance and verification. Although this difference is not new, and applies to any type of software, there are some new issues that are specific to agent systems. For instance, the nature of the specification is different for agents: because they are situated in a dynamic and (often) failure-prone environment the specification cannot require that the agent always succeed, since success may not be possible in all situations.

Based on the range of issues identified, we concluded that formal verification techniques need to be used as part of a toolkit, in combination with other techniques. In order for this to happen, however, the focus of verification research needs to broaden to include human and organisational factors, and researchers need to be mindful of the context in which verification will be used.

We then outlined a possible solution for assuring the behaviour of agent systems that hinged on:

1. Adopting an engineering stance that develops an error model (which is also used for testing and model checking), and that seeks to quantify risks and develop appropriate levels of mitigation. It is important that in developing the error model we consider broadly the errors that can occur, including human errors.
2. Using the error model to guide various activities such as testing and model checking, which are linked by building a "bridge" between them, using intermediate techniques. In doing this is it important to take care with specifications, and to capture assumptions.

We also briefly discussed the issue of programming level, and how certain errors could perhaps be avoided by, for example, designing and programming agent interactions at a higher level then message sending/receiving.

We are not the first to propose the combination of proving and testing, although the combination that we proposed does appear to be novel in its detail and working. Lowry *et al.* [292] proposed integrating testing and proving, but focused on a cost-benefit trade-off evaluation method, rather than on proposing a specific mechanism for combining testing and proving (other than doing them separately). Richardson & Clarke [369] use a formal specification to help partition the space into equivalence classes that are used in generating tests cases, they claim that "*Although this does not give total assurance in program reliability, we believe, and our evaluation supports this, that it provides strong evidence about the reliability of a program*". Dill [148] proposed a one-dimensional taxonomy which is similar in some ways to the one we use. Young & Taylor [444] also propose a taxonomy, and discuss using this to guide combining different techniques. They focus on approaches that would be regarded as testing rather than proving (e.g. testing, symbolic execution, and other analyses), but do briefly discuss proving (in section 7). Owen [330] is more recent, and considers the combination of tools from a pragmatic and empirical perspective. He uses a specification that has been randomly seeded with faults, and finds that many of the faults are not found by all of the tools, and that whilst running time varies, the vast majority of the faults are easy for at least one of the tools. He then proposes a detailed process for using the tools in combination to augment each other.

My hope is that this paper will perhaps encourage researchers working on agent verification techniques to reconsider their assumptions and scope, and in particular, to consider where the specification (often just assumed to be given as an input in some logic) comes from, what form it might have, what it might fail to capture, and how the specification form and notation might need to change to expand its scope (for instance to capture a social context).

As discussed in the previous sections, I see the key challenge as being how to integrate a range of techniques for obtaining assurance in such a way that allows them to strengthen and complement each other. For instance, finding a way of using testing and proving together so that we benefit from the ability of proving to consider all possibilities, and from the ability of testing to (sometimes) detect implicit assumptions. Whilst section 12.8 presented an approach for doing this, the approach is still not well developed, and more work is required, both on this specific approach, and on other approaches.

The key area for future work is thus to explore methods for obtaining assurance, such as the one presented in sections 12.7 and 12.8. Key considerations include how to capture the environment; how to develop an error model using techniques such as hazard analysis, event and fault trees; and how to include human and societal aspects in these models (by building on the work discussed in section 12.7.1).

A secondary area for work concerns approaches for specifying and implementing agent interactions. As discussed in section 12.7.2, approaches that work at the level of message sending are low-level and error-prone. A number of alternative approaches have been proposed in the literature, but these need to be assessed from an assurance perspective. Are these approaches really less error-prone? And are they easier to verify?

Some specific, more immediate, directions for future work include the following:

- Exploring the ideas of the previous section in the context of a larger, more realistic, case study.
- Determining whether (as discussed at the end of section 12.5.1) the corrected specification is harder to verify, and if so, how much harder?

- Looking at adapting ideas from the testing literature for agents, for example how might metamorphic testing [448] be used to test agent systems?
- In the previous section we argued that the various activities allowed us to have "some confidence" about a range of results. How much confidence? Can this be assessed? How?
- Looking at adapting ideas from the dynamical systems literature (e.g. [29]).

Acknowledgements I would like to thank Stephen Cranefield for discussions relating to this work, and for comments on a draft of this chapter. I would also like to thank the anonymous reviewers for their comments which were helpful in improving this paper.