

From Formal Specification to Model Checking of MAS Using CSP-Z and SPIN

Ahmed Hadj Kacem¹ and Najla Hadj Kacem²

¹ Faculté des Sciences Économiques et de Gestion de Sfax
Laboratory MIRACL
B.P.1088, 3018 Sfax, Tunisia
ahmed@fsegs.rnu.tn

² École Nationale d'Ingénieurs de Sfax
Research Unit ReDCAD
B.P.W., 3038 Sfax, Tunisia
najla.hadjkacem@isimsf.rnu.tn

Abstract. *As a result of the increasingly predominance of agent technology, there has been a lot of interest in developing agent-based methodologies. In particular, formal methodologies have recently received the attention of the agent community. One of the key features of these methodologies is their emphasis on the use of formal methods as a means to trust multiagent systems (MAS) to behave as expected. The main purpose of this paper is to extend the development process of a formal approach for designing agent-based applications, called ForMAAD. The effort expended in the added phase is concentrated on two tasks: formally specify MAS to provide a more concrete specification, and verify that the specified system fulfils correctness properties. The adoption of formal techniques from the concurrency theory is founded on the view of MAS as a computational organization of concurrent problem-solving entities.*

Keywords: *agent-oriented software engineering, ForMAAD, specification, CSP-Z, model checking, SPIN.*

Received: May 1, 2006 | **Revised:** June 30, 2006 | **Accepted:** September 30, 2006

1 Introduction

Research into methodologies for agent-based systems has dealt over the past years with analysing, designing and implementing software systems. The focus has to guide designers through the software life cycle from problem description to implementation. Currently, as the designers cope with increasingly complex and large-scale applications, the delivery of bug-free systems turns out to be hard to make with standard techniques. To tackle this challenge, formal methods are perceived as an appropriate way of increasing confidence in software engineering. Formal methods *can* be useful in developing agent-based systems, in particular when critical applications are being developed, when prototyping agent-based systems at a high-level and when developing complex cooperating systems [1].

ForMAAD (Formal Method For Agent-based Application Design) [2] is considered to be among this new trend of formal MAS methodologies. Designing agents society in a rigorous and incremental way based on stepwise refinements is the basic goal of ForMAAD. Along this process, design issues include

organization structure, assignment of functionalities (roles) to computational entities (agents) and high-level interactions between these entities. Essentially, the key idea for understanding and mastering the inherent complexity of MAS is to build system design by means of successive refinements. Two complementary levels are supported: individual and collective levels that are closely related to inter and intra-agents aspects. System comprehension (understanding system structure and behaviour) is the mainstream topic in ForMAAD specification design using Temporal Z. As a result, several functional properties of the system are not well-specified, and concurrency, one of the most important advantages of multiagent solutions, is not exploited. This means that the obtained system design cannot readily be employed for implementation. A large gap exists between the design and implementation. This paper seeks to address this gap by studying how it can be bridged.

The objective of this work is twofold. Firstly, we aim at extending the formal development process with a phase that, based on the pre-established design of ForMAAD, provides a more concrete speci-

fication describing and reasoning about concurrent problem-solving entities. Our primary tool in this phase is the specification language CSP-Z [3]. As its name indicates, CSP-Z is a combination of the process algebra CSP and the model-based notation Z, in such a way it models simultaneously dynamic and static aspects of agents in terms of their knowledge and interactions. Having established system correctness with mathematical rigour of CSP-Z, we secondly aim at checking whether the system specification satisfies correctness properties. Until recently, the verification of CSP-Z specifications is well supported by the model checking technique using FDR [3]. But for lack of this tool, we had to seek for the availability of an adequate tool support. The selected one is the model checker SPIN (Simple ProMela INterpreter) [4]. Its selection is motivated by two facts. First, it has been successfully applied to trace logical design errors in distributed systems, such as operating systems, data communication protocols, concurrent algorithms, etc. [4]. Second, its input language is CSP-like in the sense that it shares many features with CSP. Two notations are supported by SPIN: the modelling language called Promela (Process Meta Language) to build verification models and Linear Temporal Logic (LTL) [5] to express correctness properties. The Z/Eves theorem prover [6] is used as well for syntax and type checking of Z specifications.

The remainder of the present paper is structured as follows. Section 2 begins by discussing through related work the appropriateness of formal techniques from concurrency theory to MAS. Section 3 gives an overview on CSP-Z and presents the basic usage modes of SPIN, namely, as a simulator and as a verifier. Then, section 4 proposes the essential features of the translation of CSP-Z into Promela to performing the model checking of correctness properties. To emphasize our proposals, section 5 details a case study about air traffic control. Finally, the last section contains a short conclusion and tracks for future works.

2 Related work

Concurrent and distributed systems have long been recognized as one of the most complex classes of computer system to design and implement. A great deal of research effort has been devoted to understanding this complexity, and to developing formalisms and tools that enable a developer to manage it [7]. By their nature, multiagent systems tend to be multi-threaded. The problems inherent in multi-threaded systems do not go away, just because the adoption of an agent-based approach [8]. The most known problems such as deadlock, livelock and mutual exclusion

also apply to agent-based systems whenever agents interact in unexpected ways.

From this view, a number of works in MAS research investigate a range of process-oriented formal methods, such as CCS [9][10] and Petri nets [11][12][13]. Although these works show the feasibility of the approach, they suffer from the limitation of poor capacity to capture only behavioural aspects. Obviously, all the aspects of MAS can neither be specified nor verified with only one formalism. As a result, the promising tendency towards *multi-aspect formalisms* receives a lot of attention of the agent community. There is a trade-off between the aspects to which the specification applies and the joint use of formal methods. For example, integrations of process algebras with algebraic specification languages are more suitable to capture static and dynamic aspects of MAS. To this end, formalisms such as LOTOS [14], π -calculus [15] and CO-OPN [16] are investigated. In case of real-time MAS, additional aspect of time is specified using TCOZ [17]. Much of these research works are concerned with the specification of agents without performing verification. Therefore, the correctness of the system cannot be proved. It is recently that model checking techniques have begun to find a significant audience in the multi-agent systems community [18]. The works investigating SMV [19][20] and UPPAAL [21] are among the proposals for applying model checkers of concurrent systems to MAS.

While solutions in several works are to resort to the existing formal technologies of concurrent systems, other ones propose formalisms for specifying MAS. But to perform verification, they exploit existing tools. For instance, in [22] and [18], respectively MABLE and AgentSpeak(F) languages are introduced to specify MAS. Both works translate their specifications into Promela to perform model checking with SPIN. The agents are specified there by predicates. The model checking deals with the temporal verification of these predicates.

Of course the adoption of formal techniques from the concurrency theory as a means for specifying and verifying MAS is not new. Nevertheless, there are three contributions in our work. First, it's worth pointing out that the common problem with all the works mentioned previously consists in the lack of agent-oriented analysis and design phases. These phases are critical in the development process. In ForMAAD they give rise to defining the roles in the organization, showing how these roles interact with one another. Once defined, the roles will be aggregated into agent-instances. Second, while the resulting design specification in Temporal Z provides a convenient starting point for our concern, it is so abstract that some important aspects are omitted.

In order to build a more concrete specification, we make use CSP-Z to rigorously reason about system software. Third, as we will show, one of the main problems that our application of model checking is intended to address is the independencies between agents rather than the agent internal states. We show how abstracting these internal executions can reduce the costly and time consuming verification effort.

3 Overview on used formal language and tool support

3.1 CSP-Z

Among the multiple aspects of concurrent systems and MAS too (time, behaviour, data, mobility, etc.), data-oriented and behavioural aspects are often the most considered. Such aspects can be modelled simultaneously with a combination of suitable formalisms such as CSP-Z. By combining CSP jointly with Z, CSP-Z offers good legibility and good expressiveness. Furthermore, it favours the reuse of Z specifications pre-established during the design phase of ForMAAD as well as existing tools like Z/Eves.

The basic structure of a process specification in CSP-Z is encapsulated between two keywords *spec* and *end_spec* followed by the process identifier. Its scope consists of an interface and two complementary parts (CSP part and Z part). While CSP part is used to model process interaction, Z part is used to model data types, state and operations that define the state change caused by each CSP event. Z operation schemas have their names from the channel names prefixed by the keyword *com_*. When CSP, called *main formalism*, performs an event *e*, the corresponding Z operation *com_e* is executed unless its pre-condition holds.

Within the framework of MAS, the most obvious solution is to assign every agent to a process. Generally a CSP-Z specification of an agent has the following form.

```
spec AgentId
  Interface ; CSP Part ; Z Part
end_spec AgentId
```

When the agent-processes are placed to evolve simultaneously, the behavioural description of the global system will be introduced with CSP using the parallel composition (\parallel) of processes. Within the framework of MAS, if the environment is coupled with agents then it should be considered as a process brought together with agents to interact with each other. A specification of MAS has the following form:

```
MAS_Id=AgentId1  $\parallel$ ... $\parallel$  AgentIdn  $\parallel$  EnvironmentId
```

3.2 SPIN/Promela

As seen before, MAS have the same class of problems that arise when building concurrent systems such as deadlock, livelock, starvation, race condition, etc.. Being developed to report on such problems, SPIN performs simulation and model checking on system verification models written in a high-level language called Promela. We choose Promela because it provides all necessary aspects: parallel and asynchronous composition of concurrent processes, non-deterministic and guarded control structures, sending and receiving primitives and communication channels. The interactions between processes are modelled either by synchronous (e.g, rendezvous) or asynchronous (e.g, buffered) communication channels, or by global variable sharing.

The three basic types of objects in Promela model are processes (*proctype*), message channels (*chan*) and data objects. The statement execution is conditional on its enabledness or executability. All statements are either executable or blocked. Blocking is a primary mechanism to enforce synchronizing communication between processes. The sending or receiving process blocks until the system transfers the message. To model the global system process, the initialization process *init* is often used to introduce the *main* of the code. It prepares the initial state of the system by initializing global variables and instantiating the appropriate processes. It should be employed to activate simultaneously (*run*) process-instances of agents and possibly the environment.

Given a Promela model, SPIN performs on it two complementary verification techniques: the simulation and the model checking. By means of simulation, SPIN shows how well the system reacts to certain scenarios. Simulation cannot easily explore all of the possible scenarios and therefore subtle errors can remain undiscovered. Because of the additional advantage over simulation of exploring all possible scenarios, the model checking technique has the potential to reveal serious design problems in the model.

Correctness properties can be specified as system invariants using assertions, as specialized states using *end-*, *progress-* and *accept-state* or as temporal formulas. SPIN reports on correctness violations of these properties by checking for the existence of execution paths that (i) abort through an *assert*, (ii) end in an invalid *end-state*, (iii) avoid cycling through certain *progress-state*, (iv) cycle through an *accept-state*, and (v) violate temporal claim. Once a correctness property is violated, SPIN provides a counterexample that shows through a simulation how the model can reach undesirable state.

4 Syntactic-directed translation of CSP-Z specification into Promela model

In this section, we propose the essential features of the translation from CSP-Z to Promela. Our concern here is to emphasize the importance of proving with Promela that inter-agent dependencies cannot result in unexpected interactions.

A formal CSP-Z specification is a primary abstract representation of the system. However when we turn from the specification to the verification stage with SPIN, verification systems do have physical limitations that are set by problem size, machine memory size, and the maximum runtime that the user is willing, or able, to endure [4]. For the sake of the applicability of model checking (without state space explosion), these constraints must be taken into account. A way to deal with state space is to make the verification model more general using the technique of abstraction [23]. It is a technique to build a more abstract model while still preserving properties of interest. Our issue is of the appropriate level of abstraction.

According to our experiment with a first case study of prey-predators [24], when building an abstract model, we are faced with the following dilemma: if the level of abstraction is too low then the state space will be large and verification will be difficult. However, if the level of abstraction is too high, the representation will be obscured. Choosing the most appropriate level of abstraction for our verification model is not thus trivial.

On the basis that the emphasis in Promela models is placed on the coordination and synchronization aspects of a system and not on its computational aspects [25], we propose a translation method of CSP-Z specification into Promela model. The idea is the building of a verification model that's guided by the deduction of the desired properties to be verified. If the property is of interaction in the sense that is global and depends on the behaviour of all processes, it will be preserved. However, if a property is local to a process then it will be omitted. For example, local computations and data dependencies are irrelevant for the verification. There are even no reals, floats or pointers in Promela for the simple reason models are building to prove coordination and not computation.

Taking the view of agents, the problem is not related to the internal states of agents, but rather to the independencies between agents. In order for agents to deliver the overall functionality, they need to interact in a cooperative way to achieve a common goal, to coordinate their actions and to negotiate to solve conflicts. Central to modelling interacting agents, phenomena such as concurrency

and non-determinism can result in unexpected behaviour. The question then is how to manage these independencies.

It should be clear now that process interaction modelled by CSP is of considerable interest in the translation of CSP-Z into Promela. Intuitively there are seeming resemblances between CSP and Promela. Both support parallelism, non-determinism, communication and synchronization. Moreover, both are based on the notions of asynchronous process and message channel communication, however the nature of communication that occurs in CSP is different from that in Promela. The communication in the former is synchronous whereas it's in the latter both synchronous and asynchronous. These high-level similarities simplify considerably the translation of CSP-Z into Promela. As a result, CSP-Z agent-processes of the MAS are straightforwardly modelled as *proctypes* communicating through synchronous channels.

Nevertheless, there are some challenges related to Promela. By its high-level nature Promela's expressiveness is rather low; it offers a few basic data types and primitive type constructors. Similarly there are so few constructs in CSP. We propose the corresponding Promela constructs only for the most used ones. On the contrary, Z notation is considered to have wider constructs. But as argued before we mainly focus on interesting operations and data types. (see table 1)

Using this method of translation, a skeleton of a Promela model is generated and then has to be completed with reference to Promela features. For example, one can apply the atomicity construct to *non-blocking* statements by executing them in one indivisible step; without interleaved execution of other processes.

5 A case study: Air Traffic Control

The major concern of Air Traffic Control (ATC) systems is to ensure the safe operations of commercial and private aircraft. Their growing complexity due to the increasing number of flights, together with the pressure to avoid delays and collisions, makes the establishment of low-defect systems an enormous challenge.

An ATC system is often designed around airspace divisions; the sectors. In each sector, air traffic controllers have to coordinate efficiently the movement of air traffic to keep planes at safe distances from each other. In critical situations, the path of planes may inevitably be changed from the original flight plan to stave off bad weather or avoid a congested sector. Such situations can result in potential conflicts and have disastrous consequences. Negotiation

CSP	Promela
Communication channels of the Interface	
channel <i>ChanId</i> [<i>p1</i> : <i>T1</i> , ..., <i>pn</i> : <i>Tn</i>]	chan <i>ChanId</i> =[<i>0</i>] of { <i>t1</i> , <i>t2</i> , ..., <i>tn</i> }
channel <i>ChanId</i> [] (for signals transmission)	No signal notion in Promela, but let us assume: chan <i>ChanId</i> =[<i>0</i>] of { <i>bit</i> }
Processes	
$a \rightarrow P$	If the event <i>a</i> is associated with an operation schema <i>com_a</i> then check the executability of the pre-condition of <i>com_a</i> . Otherwise, the event is just passed.
$a1 \rightarrow P1 \mid \dots \mid an \rightarrow P2$	non-deterministic structures:
$P \square Q$ or $P \sqcap Q$	<i>if.fi</i> or <i>do..od</i>
$P \triangle Q$	{ <i>P instructions</i> } unless { <i>Q instructions</i> }
Z	Promela
State schemas	
Declaration part : Free Type Schema type Axiomatic definition	<i>mtype</i> declaration Data structure <i>typedef</i> C Macro or <i>inline</i> definition
Predicate part : invariant state	<i>assert</i> (invariant)
Operation schemas	
Pre- and post-condition	If the guard (standing for the pre-condition) is executable, the following instructions (standing for the post-condition) will be executed.

Table 1. Syntactic correspondences between CSP-Z and Promela

can be seen as a method for coordination and conflict resolution between self-interested entities that must communicate by exchanging proposals and counterproposals in order to reach an agreement.

In the agent-based solution of the work of [26], negotiating agents are assigned to planes. The scope of this solution encapsulates a negotiation model between two planes in conflict situation; both have the same altitude and are flying on two different routes. Relying on radar and environmental visual observation (standing for agent perception field), planes are able to detect and solve potential conflicts. Their negotiation involves the iterative exchange of proposals and counterproposals to change one plane's speed or altitude.

It's worth noting that this solution is a simplified prototype. How to deal with a negotiation model between agents is the main focus of this case study. The description below details our problem of ATC.

As a plane approaches a waypoint, its perception field detects incoming radar signals that inform of a plane's presence. Hence the plane, acting as the *detector*, sends its next waypoint in the direction of the other plane, acting as *detected*, in order to check whether it is a conflict. Having compared the waypoints, the *detected* answers with conflict's absence in which case the negotiation terminates, or a proposal for changing speed. Upon receipt of the proposal, the *detector* evaluates according to its capabilities whether it can accept or reject it. In the

first case, the *detected* will be informed of the conflict's resolution and the negotiation terminates. In the other case, the *detector* makes a counterproposal for changing speed. In the same way, the *detected* evaluates the counterproposal. It switches the proposal to changing altitude in case the counterproposal is rejected. The *detector* evaluates the new proposal according to its capabilities. At extreme case, if the proposal is rejected then the *detector* makes a counterproposal for changing altitude. Inevitably the *detected* has to accept the last proposal.

5.1 CSP-Z specification

We present here a CSP-Z specification of the problem. We firstly introduce the Z global declarations to be accessible by all agent-processes. Then, we show the specification of agent-processes, we consider only the plane1. Finally, we give the behavioural description of the overall system.

Z global declarations: The airspace of ATC is divided into sectors. Within each sector are highways, called routes, connected with point, called waypoints. Each waypoint is marked by a position, a name and a radius. To ensure the security of planes, only one plane may travel across a radius.

$$Pos \hat{=} [x, y : \mathbb{N} \mid x > 0 \wedge y > 0]$$

$$WayPoint \hat{=} [name : String; pos : Pos; radius : \mathbb{N}]$$

Between two waypoints, a route is divided into a sequence of parallel corridors. A corridor is defined by an altitude, a starting and an arrival point. When a corridor crosses with another route's corridor, the crossing is called a waypoint (core of our conflict situation).

$Corridor \hat{=} [alt : \mathbb{N}; wp_dp, wp_arr : WayPoint;$
 $free : Bool \mid wp_dp.pos \neq wp_arr.pos]$

For maintaining safe distances between corridors of the same route, a distance $DiffAltMax$ is given.

$| DiffAltMax : \mathbb{N}$

$Route$ $Num : \mathbb{N}$ $SeqCorr : seq Corridor$ $\forall i, j : \mathbb{N} \mid i \neq j \wedge 1 \leq i \leq \#SeqCorr \wedge 1 \leq j \leq \#SeqCorr \bullet$ $(SeqCorr(i)).alt \neq (SeqCorr(j)).alt$ $\forall i : \mathbb{N} \mid 1 \leq i \leq \#SeqCorr \bullet$ $(SeqCorr(i+1)).alt - (SeqCorr(i)).alt = DiffAltMax$

All inter-agent-processes communications are modeled by message passing channels. We present here the messages that are interpreted similarly by communicating agents.

$Message ::= SolvedConflict \mid NotConflict \mid NotChgSpeed \mid$
 $ChgAltitude$

Using message exchanges, the agents in conflict conduct negotiation to reach an agreement: one plane has to change either its speed (speed up/slow down) or altitude (go up/go down to the adjacent corridor).

$Go_up_ : \mathbb{P}(Corridor)$ $\forall corr : Corridor \bullet Go_up\ corr \Leftrightarrow corr.free = T$

Agent-processes specification: At the design phase two agents are defined, one agent-process per plane; *plane1* and *plane2*. Three roles are defined as well: *Detector*, *Negotiator* and *Solver*. Each role identifies capabilities which the plane must provide. Organizational relationships between planes are not static, they vary over time.

We consider in the following the plane1's process as a typical example of plane2's one. For the sake of clarity, we present the CSP-Z specification of the plane1 as follows.

1. The interface description

The interface is described as a set of events and includes *channels* and *local channels*. The key difference between them is the visibility: channels are visible by other processes whereas local ones are not.

Channels are uni-directional, connect a pair of processes and have a type associated with their names. For example, the processes of both planes are connected via the channel *sensepl1pl2* that passes values of type *WayPoint*, with plane1 sending message and plane2 receiving it.

channel *sensepl1pl2*: [*wp_arr1*: *WayPoint*], *sensepl2pl1*: [*wp_arr2*: *WayPoint*]
channel *notconflictpl1pl2*, *notconflictpl2pl1*: [*msg1*: *Message*]
channel *conflictpl1pl2*: [*pos1*: *Pos*, *spd1*: *Speed*], *conflictpl2pl1*: [*cur_pos2*: *Pos*, *cur_spd2*: *Speed*]
channel *solvedconflictpl1pl2*, *solvedconflictpl2pl1*: [*msg1*: *Message*]
channel *chgspeedpl1pl2*: [*pos1*: *Pos*, *spd1*: *Speed*], *chgspeedpl2pl1*: [*cur_pos2*: *Pos*, *cur_spd2*: *Speed*]
channel *notchgspeedpl1pl2*, *notchgspeedpl2pl1*: [*msg1*: *Message*]
channel *chgaltitudepl1pl2*, *chgaltitudepl2pl1*: [*msg1*: *Message*]
local_channel *detectpl2*: []
local_channel *speed_up1*, *slow_down1*, *unable_chgspd1*: []
local_channel *go_up1*, *go_down1*, *unable_chgalt1*: []

2. The CSP Part

The behavioural description of the plane1 is described as an interleaving of NORMAL (performing the process of flying), DETECTOR (the process of the plane1 if it's detector of a conflict) and DETECTED (the process of the plane1 if it's detected). The first process NORMAL is not of interest in our case, we simply assume that the process is active.

An inter-planes communication involves an initiator process and a responder process. As the communication evolves, both processes change their states. We consider this move from a state to another through a hierarchy of processes added to the main equation. Their names originate from the initiator and responder sides jointly with a number to make difference between them.

main = NORMAL ||| DETECTOR ||| DETECTED

DETECTOR = detectpl2 \rightarrow sensepl1pl2!wp_arr1 \rightarrow DD2_DR1(I)

DD2_DR1(I) = notconflictpl2pl1?msg1 \rightarrow **main** | conflictpl2pl1?cur_pos2,cur_spd2 \rightarrow DR1_DD2(II)

DR1_DD2(II) = speed_up1 \rightarrow solvedconflictpl1pl2!msg1 \rightarrow **main** \square slow_down1 \rightarrow solvedconflictpl1pl2!msg1 \rightarrow **main** \square unable_chgspd1 \rightarrow chgspeedpl1pl2!pos1,spd1 \rightarrow DD2_DR1(III)

DD2_DR1(III) = solvedconflictpl2pl1?msg1 \rightarrow **main** | notchgspeedpl2pl1?msg1 \rightarrow DR1_DD2(IV)

DR1_DD2(IV) = go_up1 \rightarrow solvedconflictpl1pl2!msg1
 \rightarrow **main** \sqcap go_down1 \rightarrow solvedconflictpl1pl2!msg1 \rightarrow
main \sqcap unable_chgalt1 \rightarrow chgaltitudepl1pl2!msg1 \rightarrow
 DD2_DR1(V)

DD2_DR1(V) = solvedconflictpl2pl1?msg1 \rightarrow **main**

DETECTED = sensepl2pl1?wp_arr2 \rightarrow DD1_DR2(I)

DD1_DR2(I) = notconflictpl1pl2!msg1 \rightarrow **main** |
 conflictpl1pl2!pos1,spd1 \rightarrow DR2_DD1(II)

DR2_DD1(II) = solvedconflictpl2pl1?msg1 \rightarrow **main** |
 chgspeedpl2pl1?cur_pos2,cur_spd2 \rightarrow DD1_DR2(III)

DD1_DR2(III) = speed_up1 \rightarrow solvedconflictpl1pl2!msg1
 \rightarrow **main** \sqcap slow_down1 \rightarrow solvedconflictpl1pl2!msg1
 \rightarrow **main** \sqcap unable_chgspd1 \rightarrow
 notchgspeedpl1pl2!msg1 \rightarrow DR2_DD1(IV)

DR2_DD1(IV) = solvedconflictpl2pl1?msg1 \rightarrow **main**
 | chgaltitudepl2pl1?msg1 \rightarrow DD1_DR2(V)

DD1_DR2(V) = go_up1 \rightarrow solvedconflictpl1pl2!msg1 \rightarrow
main \sqcap go_down1 \rightarrow solvedconflictpl1pl2!msg1 \rightarrow **main**

3. Z Part

From departure to arrival, each plane follows a pre-established flight plan. The intended airspeed, cruising altitude, route and corridor of flight are among the most essential information included in the plan with reference to our problem. As the plane takes off, it must adjust its speed and altitude to land along the appropriate corridor.

(a) State schema

Plane1
$pos1, cur_pos2 : Pos$ $spd1, cur_spd2 : Speed$ $corr1 : Corridor$ $rt1 : Route$ $path1 : seq WayPoint$ $perception1 : \mathbb{F} Pos$ $wp_arr2 : WayPoint$ $msg1 : Message$
$perception1 = \{p : Pos \mid pos1.x \leq p.x \leq pos1.x +$ $PerMinMax \wedge pos1.y \leq p.y \leq pos1.y + PerMinMax\}$
$corr1.wp_dp.pos.x \leq pos1.x \leq corr1.wp_arr.pos.x$ $corr1.wp_dp.pos.y \leq pos1.y \leq corr1.wp_arr.pos.y$
$\exists i : \mathbb{N} \mid 0 < i \leq \#rt1.SeqCorr \bullet rt1.SeqCorr(i) = corr1$
$\exists i : \mathbb{N} \mid 0 < i \leq \#path1 \bullet path1(i) = corr1.wp_dp \wedge$ $path1(i+1) = corr1.wp_arr$

Recall that *PerMinMax* is the scope of the perception field of each plane.

(b) The initialization schema

The emptiness of the plane1's perception field is our main issue in the initialization schema.

$InitPlane1 \hat{=} [Plane1' \mid perception1 = \emptyset]$

(c) Z operations

For reasons of brevity, we present below some typical Z operations.

Once the plane1 detects the plane2's presence, it informs it about the next waypoint in order to check whether it's a conflict situation.

$com_detectpl2 \hat{=} [\exists Plane1 \mid \exists pl2 : Plane2 \bullet$
 $pl2.pos2 \in perception1]$

$com_sensepl1pl2 \hat{=} [\exists Plane1; wp_arr1! : WayPoint \mid$
 $wp_arr1! = lane1.wp_arr]$

Having sent its destination, the plane1 waits until it receives either a message *NotConflict* or the plane2's position and speed.

$com_notconflictpl2pl1 \hat{=} [\Delta Plane1; msg1? : Message \mid$
 $msg1' = msg1?]$

$com_conflictpl2pl1 \hat{=} [\Delta Plane1;$
 $cur_pos2? : Pos; cur_spd2? : Speed \mid$
 $cur_pos2' = cur_pos2? \wedge cur_spd2' = cur_spd2?]$

The overall system specification: The behavioural description of the global system is introduced by the parallel composition of the system's basic entities: the plane1 and the plane2.

$ATC_SYS = Plane1 \parallel Plane2$

5.2 Verification in SPIN

We illustrate here the application of our proposed translation to the specified system. Our main concern is to verify the correctness of the system with respect to desired properties. We firstly present a part of the Promela model generated after the abstraction step, then we express the desired properties and finally we give the model checking results.

The main purpose of the designed system is to avoid collision in a waypoint. Since the satisfaction of this main property depends on whether plane1 or plane2 solves the conflict, we have to verify that the conflict resolution is done by only one plane. Verifying the property is given by verifying that the two planes reach an agreement after the negotiation so that one plane changes its speed or altitude. Of course the aim of the stated property does not concern the local executions of changing speed/altitude, but rather that iterative exchange of proposals and counterproposals meets our expectations.

progress. Progress labels are used to mark all states of communication; they claim that all states are required to be visited in at least one execution path. The correctness of the property is violated if at least one state is not visited.

SPIN has a special mode to prove absence of non-progress cycles [27]. It does so with the predefined LTL formula:

$$\diamond \square (\text{np_})$$

`np_` is global predefined, read-only variable, it is defined to be *true* in all states that are not marked as progress states, and is *false* in all other states.

Property3: "The message sequence performs as expected."

This property can be stated as "if a plane sends a message then eventually, it will receive an answer". For instance, if the plane sends its next waypoint then eventually, it waits for a reply denoting if a conflict situation occurs or not.

```

□ (p → ◇ q)
#define p (sense12==true)
#define q ((not_Conflict==true) || (Conflict==true))

```

Once the communications are verified, we can trust the agents to communicate as expected.

Model checking results: The exhaustive model checking results are summarized in table 2. For each property, we give its result, the number of states explored and the memory usage.

Property	Result	No. states	Memory (Mb)
1	true	339	2.622
2	true	257	2.622
3	true	189	2.622

Table 2. Model checking results

During the model checking, behavioural properties are of interest while data-based properties are omitted. The verification of this final kind of properties, although important, results in larger state space and afterwards in the problem of explosion. The use of non-trivial data structures (arrays, *typedef* structures) leads to higher memory usage, but the use of trivial ones (*bit*, *byte*, *bool*) requires less memory usage.

6 Conclusion

The contribution of this paper is to address the gap between the design stage of ForMAAD approach and

implementation. The focus is to build, on the basis of the resulting design specification, a more concrete system specification that fulfils correctness properties. Our effort is not in the direction of developing from scratch techniques specific to MAS. Rather, we take advantage of the research effort devoted to concurrency theory by exploiting existing tools. Of the specification formalisms, we investigate CSP-Z to make MAS specifications more concise, less ambiguous, in such a way it is easier to reason about them. Among the tool supports, SPIN is investigated to exploit (i) the simulation facilities, especially the MSC, in order to support early fault detection, and (ii) the technique of model checking to prove correctness properties. As our concern is to perform verification of desirable properties on MAS specifications, we propose a syntactic-directed translation of CSP-Z specification into Promela model. The generated model is kept as close as possible to the specification. This is because Promela is CSP-like and the emphasis is on interactions between agents rather than computational aspects.

There are many issues remaining for future work. First of all, the translation from CSP-Z to Promela has not yet been formally proved. It has to be directed by syntax as well as semantics. After proving that desirable properties are preserved, it is possible to apply a refinement process such that the specification is translated into more and more concrete representation. In the final representation, the MAS can be executed in a programming language.

References

1. Fisher, M., Müller, J., Schroeder, M., Staniford, G., Wagner, G.: Methodological foundations for agent-based systems. *Knowl. Eng. Rev.* **12**(3) (1997)
2. Regayeg, A., Hadj Kacem, A., Jmaiel, M.: Towards a formal methodology for designing multi-agent applications. In: Third German conference on Multi-Agent System TEchnologies (MATES'05). Volume 3550 of Lecture Notes in Artificial Intelligence., Koblenz, Germany, Springer-Verlag (2005) 153–164
3. Mota, A., Sampaio, A.: Model-checking CSP-Z : Strategy, tool support and industrial application. *Science of Computer Programming* **40** (2001) 59–96
4. Holzman, G.J.: The model checker SPIN. *IEEE Transactions on Software Engineering* **23**(5) (1997) 279–295
5. Pnueli, A.: The temporal logic of programs. In: FOCS. (1977) 46–57
6. Meisels, I., Saaltink, M.: The Z/EVES reference manual (1997)
7. Ben-Ari, M.: Principles of concurrent and distributed programming. Prentice-Hall (1990)

