# Incremental Development of Multi-Agent Systems in Object-Z

Graeme Smith and Kirsten Winter

*School of Information Technology and Electrical Engineering,*
*The University of Queensland, Australia*

***Abstract*—The complexity of multi-agent systems (MAS) demands a formal and incremental approach to their development. Such an approach needs to take into account issues specific to the development of MAS. In particular, methods are required for incrementally introducing agent decision-making procedures, and inter-agent negotiation mechanisms. This paper introduces an approach to modelling MAS and a definition of action refinement in Object-Z aimed at addressing these issues.**

***Keywords*-multi-agent systems, action refinement, Object-Z**

## I. Introduction

Due to the intrinsic complexity of multi-agents systems (MAS), it has been foreseen that formal methods will play an increasing and fundamental role in supporting their development [4]. To date, however, the work in this area is quite limited. Formal frameworks for categorising agents and specifying MAS have been proposed using existing state-based formal notations such as Z [6] and Object-Z [10]. However techniques for incrementally developing specifications, from abstract specifications capturing basic system functionality to more concrete specifications capturing implementation details, have not been considered. We regard such techniques as the key to coping with the complexity of MAS.

While Z and Object-Z have well defined notions of data refinement supporting incremental development [16], [5], these notions are not ideal for dealing with the specific issues that arise in the development of MAS. The complexity of individual agents largely arises from their "intelligent" decision-making procedures. These procedures determine whether an agent performs a particular action in a given context. At a high-level of abstraction, we would like to ignore such procedures by leaving the choice of actions nondeterministic. Adding them at a lower level of abstraction would then require that the occurrence of certain actions be restricted. Restricting when an action can occur, however, is not supported by data refinement in either Z or Object-Z: if an action can occur in a given situation in the abstract specification, it must also be able to occur in the same situation in the concrete specification.

Additionally, the complexity of a MAS arises from the interactions between its agents. These interactions capture negotiations between agents which must cooperate to perform tasks. At a high-level of abstraction we would like to ignore such negotiations, focussing instead on their outcomes. Adding the negotiations at a lower level of abstraction would then require the addition of further actions modelling the sending and receiving of messages. Adding actions, however, is not supported by standard data refinement in either Z or Object-Z: there must be exactly one concrete action corresponding to each abstract action.

These issues with refinement can be overcome using action refinement as defined by Back for action systems [2]. Action refinement allows the occurrence of actions to be restricted, provided this doesn't result in the overall system deadlocking, and new actions to be added to a specification, provided they do not introduce additional transitions of the *global*, *i.e.*, externally observable, system state. In this paper, we define a notion of action refinement for Object-Z and demonstrate its utility on a case study from the MAS literature. We choose to use Object-Z, rather than action systems or other notations such as Event-B [1] which already support action refinement, due to its explicit support for classes and objects which facilitate the specification of agents and MAS.

In Section II, we overview Object-Z and our approach to using it to model MAS. We also present our definition of action refinement for Object-Z building on that of action systems. In Section III, we present an abstract specification of a multi-agent system proposed by Excelente-Toledo and Jennings [8]. We then use action refinement to introduce an agent decision-making procedure and agent negotiations in Sections IV and V respectively. We conclude in Section VI by considering future work.

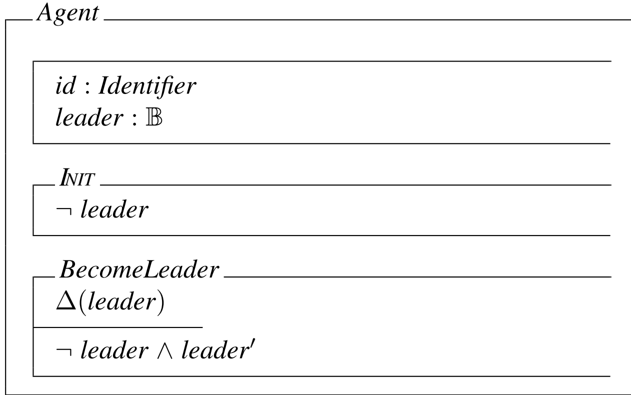
## II. Action refinement in Object-Z

### A. Object-Z

Object-Z [12] is an object-oriented extension of Z [15], [16], a state-based formal specification language in which system states, initial states and operations are modelled by schemas comprising a set of variable declarations constrained by a predicate. A class in Object-Z encapsulates a state schema, and associated initial state schema, with all the operation schemas which may change its variables. Classes have been shown useful for modelling the behaviour of agents in multi-agent systems (MAS) [10], [14]. For example, consider the following specification of a simple

 ©IEEE Computer Society Press, 2012

agent which has an identifier and may become leader of a group of similar agents.



The class has two state variables, *id* of a type *Identifier* and *leader* of type Boolean. Initially, the agent is not a leader and the operation *BecomeLeader* allows it to become a leader. The lower part of the operation is a predicate describing the operation's effect in terms of the values of the state variables before and after the operation; those after the operation are decorated with a prime, *e.g.*, $leader'$. When the predicate cannot be satisfied, *e.g.*, because *leader* is already true, then the operation cannot occur. This is in contrast to Z operations which can occur at any time but may have undefined behaviour [16]. The upper part of the operation contains a $\Delta$-list indicating which variables the operation may change; all other variables are implicitly unchanged, *e.g.*, $id' = id$ in the above operation. This part of the operation may also include declarations of variables local to the operation (such as inputs and outputs).

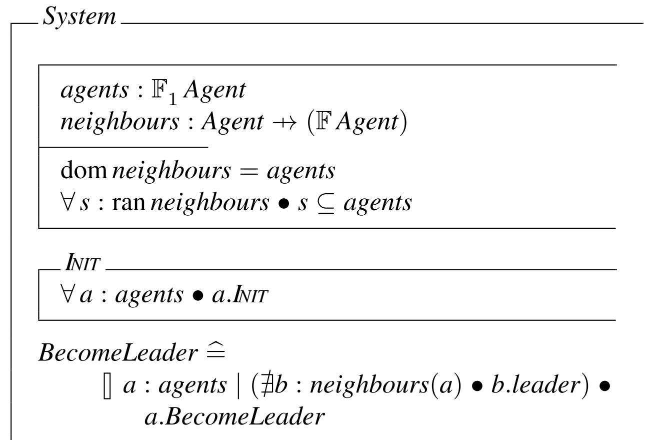
In this example, we might want our agent to become a leader only when no other agent in its neighbourhood is a leader. Hence, we need to constrain when the *BecomeLeader* action can occur. In the interest of keeping the specification abstract (and hence easy to understand and reason about), we can specify such inter-agent constraints (and environmental constraints on the agent in general) in another class describing the entire MAS.

In this paper, we adopt conventions for specifying the class describing a MAS. Its state variables should include one or more variables representing the agents and zero or more variables representing other aspects of the environment in which the agents operate. Each operation is of the form $Select \bullet AgentOp \wedge EnvironmentOp$ where *Select* chooses one or more agents and/or instances of entities in the environment, *AgentOp* describes how the agents change, and *EnvironmentOp* describes how the environment changes. The $\bullet$ after *Select* is an Object-Z operator which introduces the selected instances into the scope of the operations *AgentOp* and *EnvironmentOp*. The conjunction between the two latter operations requires both of their effects to be true. When there is no change to either the agents or the environment, the associated operation need not appear.

The *Select* part of the operation generally has two forms:

1. $[]\ x : X;\ y : Y \mid p(x, y)$ when the operations following the $\bullet$ are to be performed for a particular assignment of values to variables $x$ and $y$ (representing agents and/or environmental entities) satisfying the predicate $p(x, y)$, or
2. $\wedge\ x : X;\ y : Y \mid p(x, y)$ when the operations are to be performed concurrently for all values of $x$ and $y$ satisfying $p(x, y)$.

As an example, consider the following MAS specification which ensures an agent only becomes a leader when none of its neighbours is already a leader. ($\mathbb{F}\, X$ denotes a finite set of elements of type $X$, and $\mathbb{F}_1\, X$ denotes a non-empty, finite set of elements of type $X$. $f : X \pfun Y$ denotes a partial function from type $X$ to type $Y$, $\mathrm{dom} f$ the domain of such a function, and $\mathrm{ran} f$ its range.)



The dot notation familiar from object orientation is used to reference variables and the initial condition of class instances, and to apply operations to them. The initial state of *System* states that all agents are in their initial states, *i.e.*, are not leaders. The operation *BecomeLeader* states that a single agent $a$ becomes leader provided that none of its neighbours are already leaders.

### B. Action refinement

Data refinement for Object-Z has been defined by Derrick and Boiten [5]. They provide two sets of simulation rules which together are *complete*, *i.e.*, can be used to verify any data refinement between Object-Z specifications. One set of rules (the downward, or forward, simulation rules) does not allow an operation's *guard*, *i.e.*, when it can occur, to be either weakened or strengthened. The other set of rules (the upward, or backwards, simulation rules) allows the guard of an operation to be weakened, but not strengthened. Neither set of rules allows the replacement of a single operation by a set of operations.

As discussed in the introduction, such a notion of data refinement is not suited to the incremental development of

MAS. Derrick and Boiten [5] also define a notion of *non-atomic refinement* for Object-Z which allows an abstract operation to be refined to a sequence of concrete ones. It is not ideal for our purposes, however, as it does not allow guards to be strengthened. We therefore base our approach on the simulation rules for action refinement in action systems by Back and von Wright [3] . Here we consider the forward simulation rules only, and adapt them for Object-Z. The backwards simulation rules could be similarly adapted.

An action system has a state comprising global, *i.e.*, observable, and local variables, an initialisation condition, and a set of actions. The actions have guards which determine when they are enabled but, unlike Object-Z, the guard being enabled does not guarantee the action's definition can be satisfied. This is instead guaranteed by the action's *precondition*. If an action is enabled in a given state but its precondition is not satisfied, it is said to *abort*. A state in which an action can abort is called an *aborting state*. Action systems behave by repeatedly executing enabled actions until none are enabled, or an enabled action *aborts*. A state in which no actions are enabled is called a *terminating state*.

In order to prove refinement using the simulation rules, the specifier needs to select some of the actions to be *stuttering actions*. A stuttering action must leave the global variables unchanged. All other actions, whether or not they change the global variables, are called *change actions*. For an abstract action system $A$ and a concrete action system $C$ whose states are related by a retrieve relation $R$, the forward simulation rules are then:

**Initialisation:** Any initialisation followed by stuttering actions in C simulates (via $R$) initialisation followed by stuttering actions in A.

**Forward simulation:** Any change action in C followed by stuttering actions simulates some change action in A followed by stuttering actions, or begins from a state related (by $R$) to an aborting state of A.

**Abort:** Any aborting state in C is related to aborting states in A.

**Termination:** Any terminating state in C is related to terminating or aborting states in A.

**Infinite stuttering:** Any state in C from which infinite stuttering is possible, *i.e.*, an infinite sequence of stuttering actions can occur, is related to states in A which are either aborting or from which infinite stuttering is possible.

For Object-Z, there are no aborting states (since the guard of an operation guarantees that the operation's definition can be satisfied). It has been suggested that Object-Z be extended to include both guards and preconditions to allow abstraction from exceptional behaviour (which is modelled, at a high level of abstraction, by the system aborting) [9], [11]. In this paper, however, we use standard Object-Z. Hence, the above rules can be defined (in the absence of aborting states) as follows.

Let $A$ be an Object-Z class with state schema $AState$, initial state schema $AInit$, and operations partitioned into change actions $AChange_0, \ldots, AChange_n$ and stuttering actions $AStutt_0, \ldots, AStutt_m$ for some $n, m : \mathbb{N}$. The choice of stuttering actions must be made by the specifier based on which variables they regard as being observable. Similarly, let $C$ be an Object-Z class with state schema $CState$, initial state schema $CInit$, and operations partitioned into change actions $CChange_0, \ldots, CChange_l$ and stuttering actions $CStutt_0, \ldots, CStutt_k$ for some $l, k : \mathbb{N}$.

Let $AStutt = (AStutt_0 \lor \ldots \lor AStutt_m)$ and $CStutt = (CStutt_0 \lor \ldots \lor CStutt_k)$. $A$ is refined by $C$ when there exists a retrieve relation $R$ (modelled by a Z schema as in [5]) which relates the states of $C$ to those of $A$ such that the following hold. ($A \mathbin{⨾} B$ is the sequential composition of operations $A$ and $B$, and $A^n$ is the iteration of operation $A$ $n$ times, *e.g.*, $A^3 = A \mathbin{⨾} A \mathbin{⨾} A$. $\mathrm{pre}\, A$ returns the guard of operation $A$, and $S'$ is schema $S$ with all free variables $x$ replaced by $x'$.)

**Initialisation:** Any initialisation followed by stuttering actions in C simulates initialisation followed by stuttering actions in A. (Schemas are used below as declarations and predicates as in Z [15].)

$$\forall\, CState;\ CState';\ i : \mathbb{N} \bullet CInit \land CStutt^i \Rightarrow \\ (\exists\, AState;\ AState';\ j : \mathbb{N} \bullet AInit \land AStutt^j \land R')$$

**Forward simulation:** Any change action in C followed by stuttering actions simulates some change action in A followed by stuttering actions.

$$\forall\, AState;\ CState;\ CState';\ c : 0 \mathinner{..} l;\ i : \mathbb{N} \bullet \\ R \land CChange_c \mathbin{⨾} CStutt^i \Rightarrow \\ (\exists\, AState';\ a : 0 \mathinner{..} n;\ j : \mathbb{N} \bullet \\ AChange_a \mathbin{⨾} AStutt^j \land R')$$

**Termination:** Any terminating state in C is related to terminating states in A.

$$\forall\, AState;\ CState \bullet \\ R \land \neg\ \mathrm{pre}(CChange_0 \lor \ldots \lor CChange_l \lor CStutt) \Rightarrow \\ \neg\ \mathrm{pre}(AChange_0 \lor \ldots \lor AChange_n \lor AStutt)$$

**Infinite stuttering:** Any state in C from which infinite stuttering is possible is related to states in A from which infinite stuttering is possible.

$$\forall\, AState;\ CState \bullet \\ R \land (\forall\, i : \mathbb{N} \bullet \exists\, CState' \bullet CStutt^i \Rightarrow (\mathrm{pre}\, CStutt)') \Rightarrow \\ (\forall\, j : \mathbb{N} \bullet \exists\, AState' \bullet AStutt^j \Rightarrow (\mathrm{pre}\, AStutt)')$$

## III. Abstract Specification

Excelente-Toledo and Jennings [8] have presented a framework for agents accomplishing tasks, both unilaterally and in cooperation with other agents, on a two-dimensional grid. In this framework, an agent is assigned a *specific task*

(ST) which lies at a given position on the grid. The agent must move to its ST and accomplish it. Such an agent is said to be in state *Agent-in-ST* (AiS). On the way to its ST, an agent may come across a *cooperative task* (CT). Such tasks generally require more than one agent to accomplish them. On finding a CT, the agent needs to decide whether or not it should attempt to accomplish the CT, and if so, negotiate with other agents to come to the task and cooperate in accomplishing it. An agent which decides to coordinate a CT is said to be in state *Agent-in-Charge* (AiC) and one that decides to cooperate on a CT in state *Agent-in-Cooperation* (AiCoop). After this negotiation phase, all agents which intend to move take one step synchronously.

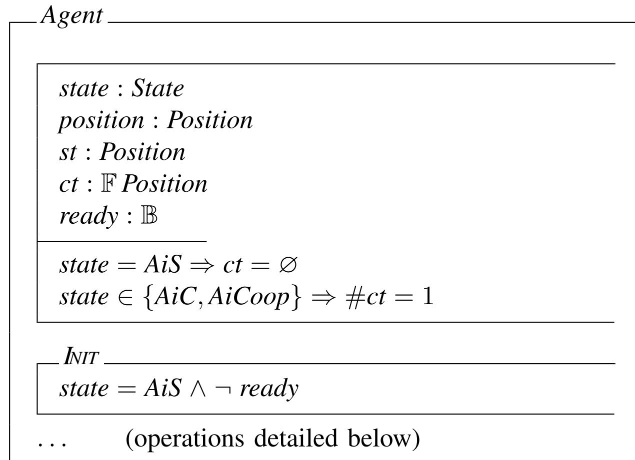
The framework is interesting as it could be used for a variety of agent-based applications, and is substantial enough to provide a non-trivial case study for our approach to formal incremental development. In this section, we formalise an abstract version of this framework using Object-Z. In our abstract specification, we ignore the details of the reward-based decision-making procedure and inter-agent negotiations.

### A. Agent specification

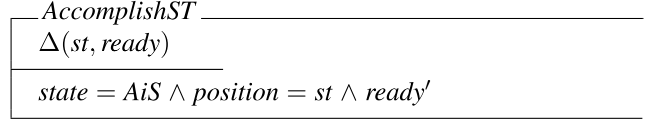
We model the grid abstractly by a set of positions, *Position*, and a function, $dist : Position \times Position \rightarrow \mathbb{N}$, which returns the distance between any two positions, *i.e.*, the minimum number of moves an agent would need to make to go from one of the positions to the other.

An agent has a state (modelled by variable *state*), a position in the grid (*position*), the position of its specific task (*st*), the position of its cooperative task (*ct*) when not in state AiS, and the agent is either ready, or is evaluating its current situation (modelled by Boolean variable *ready* which is true iff the agent has considered its current situation). The latter variable is required to ensure that all agents synchronise on their moves as required by the framework of Excelente-Toledo and Jennings [8]. Initially, an agent is in state *AiS* and is evaluating its current situation.

$State ::= AiS \mid AiC \mid AiCoop$

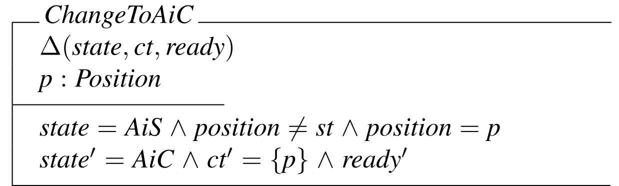



An agent in state AiS which is at the same position as its ST accomplishes the ST before becoming ready to move. When this happens, Excelente-Toledo and Jennings [8] require that a new ST is placed in the grid and allocated to the agent. We model this abstractly by including *st* in the operation's $\Delta$-list, but not constraining its post-state value in the operation's predicate.



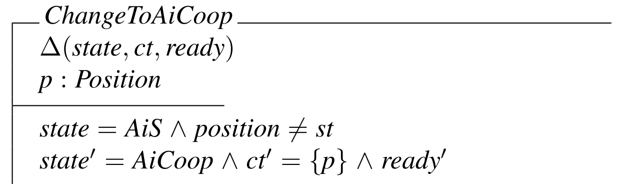

An agent in state AiS which is not at the same position as its ST may do one of three things before becoming ready.

1. If it is at the same position as a CT, it may change to state AiC. At this level of abstraction, we ignore the reasons for this decision. We will examine them in Section IV.



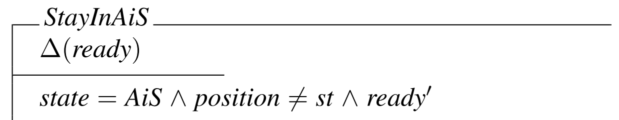

   This action is only possible when there is a CT at position *p* and it is not already being coordinated. These environmental and inter-agent constraints are captured in the *System* class below.

2. An agent may become a cooperating agent for a CT. Again we abstract away from the mechanism by which this actually occurs. This will be dealt with in Section V.




   This action is only possible when there is a CT at p which is being coordinated by another agent. Again these constraints are captured in the *System* class below.

3. In any situation, the agent may stay in state AiS. This includes the case where the agent is at the same position as a CT and decides not to coordinate the CT, and the case where another agent is in state AiC and the agent does not become a cooperating agent.




An agent which is ready to move and not already at its goal may move one step closer to its goal. This is captured

by two operations: one for agents in state AiS, and one for agents in state AiCoop. Agents in state AiC are required to remain in their position until their CT is accomplished.

An agent in state AiS may move one step closer to the position of its ST. After the agent moves, it must re-evaluate its position.

**AiSMove**

$$\Delta(position, ready)$$

$$ready \wedge state = AiS \wedge position \neq st$$
$$dist(position', st) = dist(position, st) - 1 \wedge \neg\, ready'$$

An agent in state AiCoop may move one step closer to the position of its CT. The agent does not re-evaluate its position after a move. It stays in state AiCoop until its CT is accomplished.

**AiCoopMove**

$$\Delta(position)$$

$$ready \wedge state = AiCoop \wedge position \neq ct$$
$$dist(position', ct) = dist(position, ct) - 1$$

An agent in state AiC or AiCoop which is at the same position as its CT may accomplish the CT. It then returns to state AiS and is ready to move.

**AccomplishCT**

$$\Delta(state, ct)$$

$$ct = \{position\} \wedge state' = AiS \wedge ct' = \varnothing \wedge ready'$$

This action is only possible when all agents cooperating on the CT are at the position of the CT. This inter-agent constraint is captured in the *System* class below. Also, Excelente-Toledo and Jennings require that a new CT is placed on the grid [8]. This environmental constraint is also captured in class *System*.

### B. System specification

The system comprises a finite, non-empty set of agents (*agents*), and a finite set of CTs (*cts*). We assume that there can be at most one CT at any position and so it suffices to model the set of CTs by the set of their positions.

**System**

$$agents : \mathbb{F}_1\, Agent$$
$$cts : \mathbb{F}\, Position$$

*INIT*

$$\forall\, a : agents \bullet a.\mathit{INIT}$$

$$AccomplishST \mathrel{\widehat=} [\!]\; a : agents \bullet a.AccomplishST$$
$$StayInAiS \mathrel{\widehat=} [\!]\; a : agents \bullet a.StayInAiS$$

... (other operations detailed below)

There is a system operation for each agent operation. The operations *AccomplishST* and *StayInAiS* simply promote the agent operations to system operations. The other operations involve environmental and inter-agent constraints.

*ChangeToAiC* models a single agent at the position of a CT changing to state AiC. It is required that no other agent is involved with the CT.

$$ChangeToAiC \mathrel{\widehat=} [\!]\; a : agents;\; p : cts \mid (\nexists b : agents \bullet b.ct = \{p\}) \bullet a.ChangeToAiC$$

*ChangeToAiCoop* models a single agent changing to state AiCoop. There must be another agent already involved with the CT.

$$ChangeToAiCoop \mathrel{\widehat=} [\!]\; a : agents;\; p : cts \mid (\exists\, b : agents \bullet b.ct = \{p\}) \bullet a.ChangeToAiCoop$$

*Move* models all agents which are able to move (those in state AiS or AiC and not at their goal) taking a step towards their goal. ($A \;[\!]\; B$ denotes either $A$ or $B$ occurring.)

$$Move \mathrel{\widehat=} \bigwedge a : agents \mid a.state = AiS \wedge a.position \neq a.st \vee a.state = AiCoop \wedge a.ct \neq \{a.position\} \bullet a.AiSMove \;[\!]\; a.AiCoopMove$$

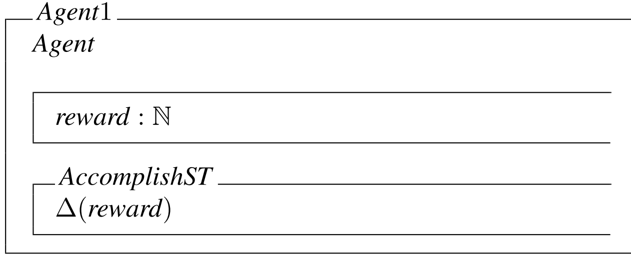
*AccomplishCT* models all agents involved with a CT accomplishing the CT. There must be at least one agent involved with the CT which is removed from *cts*. Furthermore, a new CT is added to the grid. ($[\Delta(x, y);\; z : Z \mid p(x, y)]$ is the in-line form of an operation schema.)

$$AccomplishCT \mathrel{\widehat=} [\!]\; p : cts \mid (\exists\, a : agents \bullet a.ct = \{p\}) \bullet (\bigwedge a : agents \mid a.ct = \{p\} \bullet a.AccomplishCT) \wedge [\Delta(cts) \mid \exists\, q : Position \bullet q \notin cts \setminus \{p\} \wedge cts' = (cts \setminus \{p\}) \cup \{q\}]$$

## IV. Introducing decision making

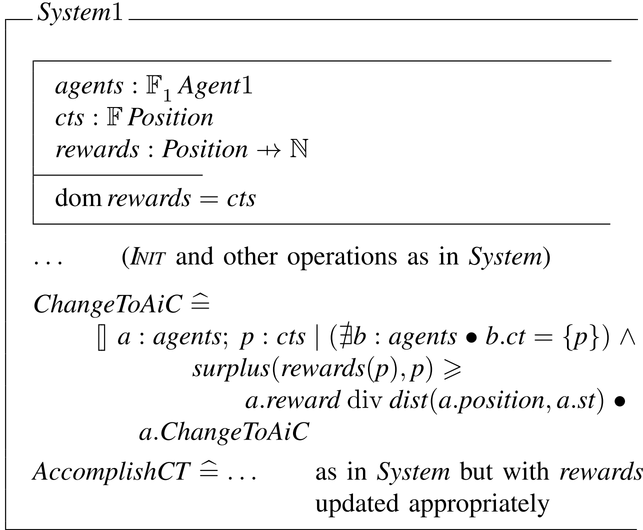
The heart of the paper by Excelente-Toledo and Jennings is the definition of the decision-making procedures which enable agents to decide on whether or not to coordinate and cooperate on CTs. These procedures are based on rewards associated with the tasks and a process of bidding and negotiating contracts for a share in the reward associated with a CT. An agent's goal is to maximise its reward per time step. In this section, we introduce the agent's decision making process in terms of rewards. The negotiation of contracts is introduced in Section V.

To model rewards we extend *Agent* (using inheritance [12]) to include a state variable *reward* denoting the reward associated with its ST. This is updated to the reward of the new ST when it accomplishes an ST (we abstract from the actual value) but is otherwise unchanged by any operation.



When an agent accomplishes its ST, it receives that task's reward. When an agent coordinates a CT, it gets a portion of the CT's reward, the rest being distributed to the cooperating agents. It needs to wait while it sets up the cooperative task, and also for the cooperating agents to arrive. Let the function $surplus : \mathbb{N} \times Position \to \mathbb{N}$ return the average reward per time-unit the agent can expect from coordinating a task with a given reward at a given position. The function can be defined as in [8].

To model the decision process we modify *System* to include a reward for each CT, and so that an agent in state AiS decides to coordinate a CT only if the expected surplus from the CT's reward is greater than or equal to the reward per time-unit of continuing to its ST.

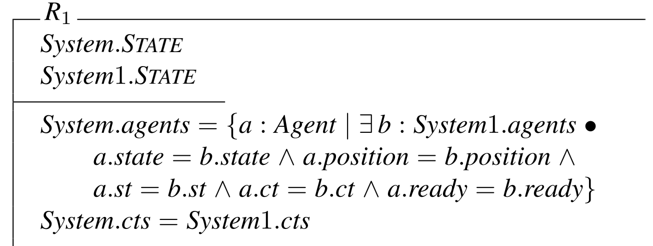


Note that the additional condition in *ChangeToAiC* does not preclude *StayInAiS* occurring for an agent for which the condition is met. This models the case where the coordinating agent is not able to find suitable cooperating agents (and so abandons the CT). The contract mechanism by which cooperating agents are recruited is the subject of Section V.

The additional condition strengthens the guard of *ChangeToAiC* and so, as discussed in Section II-B, *System*1 is not a data refinement of *System*. However, it is a sensible step in the development which can be shown using our definition of action refinement for Object-Z.

### A. Proof sketch

We assume the agent variables *state* and *ready* are local, *i.e.*, non-observable, and all other agent and system variables are global. We also choose all operations to represent change actions (*i.e.*, we have no stuttering actions). The retrieve relation $R_1$ maps each variable in *System* to the variable in *System*1 that has the same name. Using the notation $A.\textsc{State}$ to denote the state schema of class $A$ with all variables $x$ replaced by $A.x$, we define $R_1$ as follows.



*Initialisation.* Without stuttering actions the initialisation rule simplifies to the following:

$$\forall\, CState' \bullet CInit \Rightarrow (\exists\, AState' \bullet AInit \wedge R_1')$$

This obviously holds as the initialisation of *System*1 (and its agents) and *System* (and its agents) are identical on related variables (*i.e.*, all variables apart from *reward* of *Agent*1 and *rewards* of *System*1).

*Forward Simulation.* Similarly, the forward simulation rule simplifies. Both models have the same operations on related variables. The only change is a strengthening of the guard of the operation *ChangeToAiC* in *System*1. A consequence is that whenever *ChangeToAiC* is applicable in *System*1, operation *ChangeToAiC* will also be applicable in *System* and have the same effect, *i.e.*, the retrieve relation $R_1$ will relate the post-states of the two operations.

*Termination.* The termination rule holds if we can be sure that the modified operation *ChangeToAiC* in *System*1 does not introduce new deadlocking behaviour. A closer look into the guard of the operation shows that it restricts the state in which the operation is enabled to a subset of those states in which operation *StayInAiS* is enabled. As a consequence the states in which *ChangeToAiC* is enabled in *System* but not in *System*1 still satisfy the guard of *StayInAiS* in *System*1. Thus, no deadlock can occur in these states. Any other deadlocking behaviour in *System*1 will have a matching deadlocking behaviour in *System*, which satisfies the termination rule.

*Infinite Stuttering* As both system do not contain stuttering actions this rule is satisfied.

## V. Introducing negotiation

Agents wishing to coordinate a CT send a request for cooperating agents. The request comprises the (identity of the) requesting agent, and the position of the CT. Such communications can be modelled as a record type using a Z schema.

*Request*

$requester : Agent$
$position : Position$

Agents wishing to cooperate on a CT send the requesting agent a bid. The bid comprises the (identities of the) requesting and bidding agents, the distance the bidding agent is from the CT, and the reward the bidding agent requires for cooperation.

*Bid*

$requester : Agent$
$bidder : Agent$
$distance : \mathbb{N}$
$reward : \mathbb{N}$

An agent wishing to coordinate a CT sends contracts to a subset of the agents which have sent it bids. The contract comprises the (identities of the) coordinating and bidding agents, and the position of the CT.

*Contract*

$coordinator : Agent$
$bidder : Agent$
$position : Position$

The class $Agent1$ is extended with a variable *received* representing a coordinating agent's received bids, and a variable *bidded* representing the set of agents to which the agent has sent a bid.

*Agent2*

. . . other variables as in $Agent1$
$received : \mathbb{F}\, Bid$
$bidded : \mathbb{F}\, Agent$

*INIT*

. . . as in $Agent1$
$received = \varnothing \wedge bidded = \varnothing$

. . . (operations detailed below)

*AccomplishST*, *StayInAiS* and *AccomplishCT* are defined as in $Agent1$. Operations *ChangeToAiC* and *ChangeToAiCoop* are replaced by the following five operations.

An agent in state AiS which has not already considered its current position and is not at the position of its ST, but is at the same position as a CT whose surplus reward is greater than or equal to the agent's reward per time-unit for continuing towards its ST, can send a request for cooperating agents. The agent changes to state AiC. (Given an instance of a Z schema modelling a record type, the standard dot notation is used to access the values of its variables, *e.g.*, $request!.requester$ below. *self* is an implicit variable in all Object-Z classes denoting the identity of the object at hand [12].)

*SendRequest*

$\Delta(state, ct)$
$request! : Request$
$p : Position$

$state = AiS \wedge \neg\, ready \wedge position \neq st \wedge position = p$
$bidded = \varnothing \wedge request!.requester = self$
$request!.position = position \wedge state' = AiC \wedge ct' = \{p\}$

The constraints on the CT at the position of the agent are captured in the class $System2$ below.

When a request has been sent, other agents in state AiS can respond by sending bids. Such agents must have already evaluated their situation and be ready to move (*i.e.*, have performed operation *AccomplishST* or *StayInAiS*).

An agent may only send one bid per request. The bid includes the distance of the agent from the CT, and the reward the agent expects if it is chosen to cooperate. This reward is calculated by multiplying the deviation the agent would need to make from its current path to its ST by the reward per time-unit of continuing to its current ST [8].

*SendBid*

$\Delta(bidded)$
$request? : Request$
$bid! : Bid$

$state = AiS \wedge ready \wedge request?.requester \notin bidded$
$\textbf{let}\ deviation == dist(position, request?.position)$
$\quad + dist(request.position, st)$
$\quad - dist(position, st) \bullet$
$\quad bid! = (request?.requester,$
$\quad\quad self,$
$\quad\quad dist(position, request?.position),$
$\quad\quad (reward \operatorname{div} dist(position, st)) * deviation)$
$bidded' = bidded \cup \{request?.requester\}$

An agent which has sent a request receives the bids from the other agents.

*RecieveBid*

$\Delta(received)$
$bid? : Bid$

$bid? \notin received \wedge bid?.requester = self$
$received' = received \cup \{bid?\}$

An agent which has sent a request may choose the subset of bids it receives which maximises its reward. The number of bids in this subset will depend on the number of cooperating agents required for the task. The way the

subset is chosen is detailed in [8]; here we leave the choice nondeterministic (allowing any strategy to be used in the implementation). A contract is sent to each agent whose bid has been chosen. ($\{x : X \bullet f(x)\}$ denotes the set of elements $f(x)$ for each value of $x$ in $X$.)

$$
\begin{array}{l}
\textit{SendContracts} \\
\hline
\Delta(\textit{ready}, \textit{received}) \\
\textit{contracts}! : \mathbb{F}_1\, \textit{Contracts} \\
\hline
\textit{received} \neq \varnothing \\
\forall\, c : \textit{contracts}! \bullet \\
\quad c.\textit{coordinator} = \textit{self} \wedge \\
\quad c.\textit{bidder} \in \{b : \textit{received} \bullet b.\textit{bidder}\} \wedge \\
\quad c.\textit{position} = \textit{position} \\
\textit{ready}' \wedge \textit{received}' = \varnothing
\end{array}
$$

An agent whose bid has been chosen receives its contract and then changes state to AiCoop.

$$
\begin{array}{l}
\textit{ReceiveContract} \\
\hline
\Delta(\textit{state}, \textit{ct}, \textit{ready}, \textit{bidded}) \\
\textit{contract}? : \textit{Contract} \\
\hline
\textit{contract}?.\textit{coordinator} \in \textit{bidded} \\
\textit{contract}?.\textit{bidder} = \textit{self} \\
\textit{state}' = \textit{AiCoop} \wedge \textit{ct}' = \{\textit{contract}?.\textit{position}\} \wedge \textit{ready}' \\
\textit{bidded}' = \varnothing
\end{array}
$$

Note that an agent which has sent a request may also choose to abandon the CT. This would happen if the resulting bids did not allow the agent to get a large enough portion of the CT's reward (see [8]). Here we abstract from the reason (again leaving any strategy to be used in the implementation). The agent returns to state AiS and its set of received bids is deleted.

$$
\begin{array}{l}
\textit{AbandonCT} \\
\hline
\Delta(\textit{state}, \textit{ready}, \textit{received}) \\
\hline
\textit{state} = \textit{AiC} \wedge \neg\, \textit{ready} \\
\textit{state}' = \textit{AiS} \wedge \textit{ready}' \wedge \textit{received}' = \varnothing
\end{array}
$$

The following operation extends *AiSMove* to delete the agent's record of sent bids.

$$
\begin{array}{l}
\textit{AiSMove2} \\
\hline
\Delta(\textit{bidded}) \\
\textit{AiSMove} \\
\hline
\textit{bidded}' = \varnothing
\end{array}
$$

To model the asynchronous communication between agents, we add three new variables representing messages in transit to our system class. Initially, there are no messages in transit.

$$
\begin{array}{l}
\textit{System2} \\
\hline
\quad \begin{array}{l}
\textit{agents} : \mathbb{F}_1\, \textit{Agent2} \\
\textit{cts} : \mathbb{F}\, \textit{Position} \\
\textit{rewards} : \textit{Position} \nrightarrow \mathbb{N} \\
\textit{requests} : \mathbb{F}\, \textit{Request} \\
\textit{bids} : \mathbb{F}\, \textit{Bid} \\
\textit{contracts} : \mathbb{F}\, \textit{Contract} \\
\hline
\operatorname{dom} \textit{rewards} = \textit{cts}
\end{array} \\
\\
\quad \begin{array}{l}
\textsc{Init} \\
\hline
\ldots \quad \text{other constraints as in } \textit{System1} \\
\textit{requests} = \varnothing \\
\textit{bids} = \varnothing \\
\textit{contracts} = \varnothing
\end{array} \\
\\
\textit{ReceiveBid} \mathrel{\widehat{=}} [\!]\; a : \textit{agents};\ \textit{bid}? : \textit{bids} \bullet a.\textit{ReceiveBid} \\
\textit{ReceiveContract} \mathrel{\widehat{=}} [\!]\; a : \textit{agents};\ \textit{contract}? : \textit{contracts} \bullet \\
\qquad\qquad a.\textit{ReceiveContract} \\
\textit{AbandonCT} \mathrel{\widehat{=}} [\!]\; a : \textit{agents} \bullet a.\textit{AbandonCT} \\
\ldots \quad \text{(other operations detailed below)}
\end{array}
$$

The *ReceiveBid* and *ReceiveContract* operations model an agent $a$ receiving a bid or contract, respectively, from those in transit. The predicates of the agent operations ensure only those bids or contracts specifically sent to $a$ are received. The *AbandonCT* operation models an agent abandoning coordination of a CT.

The *AccomplishST* and *AccomplishCT* operations are not affected by the contract negotiation messages and are specified as in *System*1.

*SendRequest* models an agent sending a request which is added to the set of request messages in transit. This operation can occur only when the agent is at the position of a CT and there is not already an agent involved with the CT. Furthermore, it requires that the surplus the agent will receive for coordinating the CT is greater than or equal to the reward per time-unit of continuing to its ST.

$$
\begin{array}{l}
\textit{SendRequest} \mathrel{\widehat{=}} \\
\quad [\!]\; a : \textit{agents};\ p : \textit{cts} \mid (\nexists\, b : \textit{agents} \bullet b.\textit{ct} = \{p\}) \wedge \\
\qquad\qquad \textit{surplus}(\textit{rewards}(p), p) \geqslant \\
\qquad\qquad\qquad a.\textit{reward} \operatorname{div} \textit{dist}(a.\textit{position}, a.\textit{st}) \bullet \\
\qquad a.\textit{SendRequest} \wedge \\
\qquad [\Delta(\textit{requests});\ \textit{request}! : \textit{Request} \mid \\
\qquad\qquad \textit{requests}' = \textit{requests} \cup \{\textit{request}!\}]
\end{array}
$$

*SendBid* models an agent sending a bid which is added to the set of bid messages in transit. Note that a request for the bid must already be in *requests*.

$$
\begin{array}{l}
\textit{SendBid} \mathrel{\widehat{=}} [\!]\; a : \textit{agents};\ \textit{request}? : \textit{requests} \bullet \\
\quad a.\textit{SendBid} \wedge \\
\quad [\Delta(\textit{bids});\ \textit{bid}! : \textit{Bid} \mid \textit{bids}' = \textit{bids} \cup \{\textit{bid}!\}]
\end{array}
$$

*SendContract* models an agent sending a set of contracts which are added to the set of contract messages in transit.

$$SendContracts \mathrel{\widehat=} \;[\!]\; a : agents \bullet a.SendContracts \wedge \\ [\Delta(contracts);\; contracts! : \mathbb{F}_1\, Contract \mid \\ contracts' = contracts \cup contracts!]$$

An agent must coordinate a CT it comes across if this will increase its reward per time-unit. Hence, for an agent at a non-coordinated CT to stay in state AiS, the surplus from the CT must be less than the reward per time-unit of continuing to its ST. This is captured by the constraint in *StayInAiS* below.

$$StayInAiS \mathrel{\widehat=} \;[\!]\; a : agents \mid \\ a.position \in \{p : cts \mid \nexists b : agents \bullet b.ct = \{p\}\} \Rightarrow \\ surplus(rewards(a.position), a.position) < \\ a.reward \operatorname{div} dist(a.position, a.st)) \bullet \\ a.StayInAiS$$

Operation *Move* models all agents which are able to move taking a step towards their goal. All messages in transit are removed from the system in anticipation of the next round.

$$Move \mathrel{\widehat=} \bigwedge a : agents \mid a.state = AiS \wedge a.position \neq a.st \vee \\ a.state = AiCoop \wedge a.ct \neq \{a.position\} \bullet \\ a.AiSMove2 \;[\!]\; a.AiCoopMove \wedge \\ [\Delta(requests, bids, contracts) \mid \\ requests' = \varnothing \wedge bids' = \varnothing \wedge \\ contracts' = \varnothing]$$

### A. Proof sketch

We assume that all of the new agent and system variables are local, and that the new operations *SendRequest*, *SendBid*, *ReceiveBid* and *AbandonCT* are stuttering actions. All other operations are change actions. The retrieve relation $R2$ maps a *System*2 agent in state *AiC* to a *System*1 agent in state *AiS* when the agent has not sent any contracts in the current round, and to a *System*1 agent in state *AiC* when it has sent contracts. All other variables in *System*2 are mapped to the same-named variables in *System*1.

$$R_2 \\ System1.\mathrm{STATE} \\ System2.\mathrm{STATE} \\ \hline System1.agents = \{a : Agent1 \mid \exists\, b : System2.agents \bullet \\ a.reward = b.reward \wedge a.position = b.position \wedge \\ a.ready = b.ready \wedge a.st = b.st \wedge a.ct = b.ct \wedge \\ b.state = AiS \Rightarrow a.state = AiS \wedge \\ b.state = AiC \Rightarrow \\ ((\nexists c : contracts \bullet c.coordinator = b) \Rightarrow \\ a.state = AiS \wedge \\ (\exists\, c : contracts \bullet c.coordinator = b) \Rightarrow \\ a.state = AiC) \wedge \\ b.state = AiCoop \Rightarrow a.state = AiCoop\} \\ System1.cts = System2.cts \\ System1.rewards = System2.rewards$$

*Initialisation.* All common-named variables in *System*1 and *System*2 are initialised to the same values, and since each agent is initially in state AiS, *contracts* does not affect the relation between abstract and concrete states. Hence, the concrete and abstract systems are related by $R_2$ after initialisation. Furthermore, any sequence of stuttering operations in *System*2 will maintain $R_2$ as they do not alter the related variables, apart from *state* being updated to *AiC* in stuttering action *SendRequest*. However, this operation does not add any contracts to *contracts* which is intially empty and thus $R_2$ is satisfied.

*Forward Simulation.* The observable operations in *System*1 are all simulated (via $R_2$) by an observable operation in *System*2: *AccomplishST*, *StayInAiS*, and *AccomplishCT* are defined the same, *ChangeToAiC* is simulated by *SendContracts*, and *ChangeToAiCoop* by *ReceiveContract*. Furthermore as argued above, *SendRequest* is the only stuttering operation that changes a related variable by modifying *state*, but does not change *contracts*. To prove that $R_2$ is maintained by stuttering, it is sufficient to show that an agent in state AiS (the state in which *SendRequest* is enabled) is never the coordinator of a contract: something easily proved to be an invariant.

*Termination.* To prove the termination rule we again show that with the new operations in *System*2 we have not introduced deadlocking behaviour. In every possible scenario there is at least one operation enabled. Since there is at least one agent in the system ($agents : \mathbb{F}_1\, Agent2$), we reason about enabledness of the operations in the following way.

- If an agent is in state *AiS* and $\neg ready$ then either *AccomplishST*, *StayInAiS* or *SendRequest* is enabled. If all agents that are in state *AiS* are also *ready* then *Move* is enabled (since agents in state *AiCoop* are always *ready* and hence able to move).
- If none of the agents is in state *AiS* then any agent that is in state *AiCoop* and has not reached its goal *ct* can perform operation *AiCoopMove*. Hence, *Move* is enabled. If all agents in state *AiCoop* have reached their *ct* then *AccomplishCT* is enabled (since agents in state *AiC* are already at the position of their *ct*).
- If no agent is in state *AiS* or *AiCoop* then any agent in state *AiC* can perform the operation *AccomplishCT* (since it must be at the position of the *ct* to become a coordinator).

This shows that in every possible situation there is at least one operation enabled for at least one agent and thus no deadlock can occur.

*Infinite Stuttering* We show that *System*2 cannot perform an infinite sequence of stuttering actions which in turn proves this condition. Our argument is based on the fact that the system comprises only a *finite* number of agents ($agents : \mathbb{F}_1\, Agent2$). We consider each stuttering action in isolation:

- *SendRequest* can only happen once per agent as the operation invalidates its own precondition ($state = AiS$).
- *SendBid* can only happen once per agent for each request as is enforced by the precondition $request?.requester \notin bidded$. As there are only a finite number of requests (produced by a finite number of agents performing *SendRequest*) we know that this operation only occurs finitely often.
- *ReceiveBid* occurs once per sent bid as is enforced by the precondition $bid? \notin received$. With a finite number of bids this operation can occur only finitely often.
- *AbandonCT* can only occur once per agent for any CT since *ready* becomes true. This disables *SendRequest* which is the only operation which places the agent in state AiC, required for *AbandonCT*.

## VI. Conclusion

This paper has presented a formal approach to the modelling and incremental development of multi-agent systems (MAS) using Object-Z. Specifically, it has introduced a notion of action refinement for Object-Z allowing agent decision-making procedures and inter-agent negotiation mechanisms to be introduced via refinement steps. The approach was illustrated on a case study from the MAS literature.

The definition of action refinement was based on that of Back's action systems, but simplified due to the lack of aborting states in Object-Z. Such states allow abstraction from exceptional behaviour in action systems, as well as in other formalism such as Z. We plan to extend Object-Z to allow the specification of systems with aborting states to allow both for more abstract specification and a more flexible notion of action refinement.

We also plan to extend Object-Z to deal with fairness. While not an issue with the case study in this paper, fairness is often required when modelling distributed systems such as MAS to capture the fact that actions on one part of the system are not continually 'on hold' in order to let actions on other parts of the system occur. In particular, without fairness concrete specifications introducing inter-agent negotiations are prone to introducing infinite stuttering.

Finally, we would also like to explore the use of our approach in moving from abstract specifications of global system behaviour to specifications in terms of local agent behaviour. The ability to introduce additional actions (corresponding to local agent and inter-agent actions) has been shown to be well suited to this task [13], [7].

## VII. Acknowledgments

The authors would like to thank Jeff Sanders for many insightful discussions on the ideas that have led to this paper. This work is supported by Australian Research Council (ARC) Discovery Grant DP110101211.

## References

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

[2] R. J. R. Back. Refinement of parallel and reactive programs. Technical Report Caltech-CS-TR-92-23, Computer Science Department, California Institute of Technology, 1992.

[3] R. J. R. Back and J. von Wright. Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *Concurrency Theory (CONCUR '94)*, volume 836 of *LNCS*, pages 367–384. Springer-Verlag, 1994.

[4] L. Cernuzzia, M. Cossentino, and F. Zambonelli. Process models for agent-based development. *Engineering Applications of Artificial Intelligence*, 18:205–222, 2005.

[5] J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Springer-Verlag, 2001.

[6] M. d'Inverno and M. Luck. Development and application of a formal agent framework. In *Proceedings of the First IEEE International Conference on Formal Engineering Methods*, pages 222–231. IEEE Press, 1997.

[7] S. Eder and G. Smith. An approach to formal verification of free-flight separation. In *Self-Organising and Self-Adaptive Systems Workshop (SASOW 2010)*, pages 166–171. IEEE Computer Society Press, 2010.

[8] C. B. Excelente-Toledo and N. R. Jennings. The dynamic selection of coordination mechanisms. *Autonomous Agents and Multi-Agent Systems*, 9:55–85, 2004.

[9] C. Fischer. CSP-OZ - a combination of CSP and Object-Z. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, pages 423–438. Chapman & Hall, 1997.

[10] P. Gruer, V. Hilaire, A. Koukam, and K. Cetnarowicz. A formal framework for multi-agent systems analysis and design. *Expert System Applications*, 23(4):349–355, 2002.

[11] B. Mahony and J. Dong. Blending Object-Z and Timed CSP: An introduction to TCOZ. In *20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.

[12] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[13] G. Smith and J. W. Sanders. Formal development of self-organising systems. In *International Conference on Autonomic and Trusted Computing (ATC'09)*, volume 5586 of *LNCS*, pages 90–104. Springer-Verlag, 2009.

[14] G. Smith, J. W. Sanders, and K. Winter. Reasoning about adaptivity of agents and multi-agent systems. In *International Conference on Engineering of Complex Computer Systems (ICECCS 2012)*. IEEE Computer Society Press, 2012.

[15] J. M. Spivey. *The Z Notation: a reference manual, second edition*. Prentice-Hall International, 1992.

[16] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, 1996.