# A Formal Development Approach for Self-Organising Systems

Qin Li and Graeme Smith
*School of Information Technology and Electrical Engineering*
*The University of Queensland*
*Brisbane, Australia*
*qli@sei.ecnu.edu.cn; smith@itee.uq.edu.au*

***Abstract*—Self-organising systems are distributed systems which achieve an ordered global state without centralised control. They include adaptive sensor networks, swarm robotic systems and mobile *ad-hoc* networks. Designing such systems is difficult and often based on a trial-and-error approach. In this paper, we provide an approach which is both systematic and formal. Our approach builds on the formalism of Object-Z and the refinement approach of action systems. It follows an intuitive approach to development which breaks a refinement proof into three steps which the designer may iterate through on the way to the final design.**

***Keywords*-self-organising systems; Object-Z; refinement; Unifying Theories of Programming; guarded design calculus**

## I. Introduction

A self-organising system is a distributed system without centralised control that exhibits one or more self-$*$ behaviours [1]: self-configuration, self-adaptivity, self-healing etc. A characteristic of each of these self-$*$ behaviours is the achievement of a system state with some desired property. Once the state is reached the system remains in it until some *external*, *i.e.*, non-system, action renders it undesirable. This observation provides the basis for the formal design approach proposed in [2], applied in [3] and formalised in this paper. Rather than specifying the ongoing reactive behaviour of a system, we specify a system which, starting from an initial state, reaches a desired state and then terminates. This simpler model of behaviour is sufficient for verifying that a particular design produces a desired goal under certain conditions (captured by the initial state).

It allows us to capture the system at the highest level of abstraction by a single action which achieves the desired state. This action is then, through a series of refinement steps, divided into a sequence of actions representing agent interactions, resulting in a distributed design. This type of refinement is comparable to action refinement as proposed by Back for action systems [4], or non-atomic refinement proposed by Derrick and Boiten for Z and Object-Z [5]. Hence, our first approach for formal development of such systems involved defining action refinement for Object-Z [6]. Object-Z was chosen due to its expressivity and support for structuring [7], but as in this paper, was only a vehicle to explore ideas. Other notations, perhaps those better supported by tools, could also be utilised. Action refinement was chosen over non-atomic refinement due to its additional support for the strengthening of guards of individual actions (allowing non-determinism between the choice of actions to be resolved [6]).

In this paper, we extend this previous work in 3 ways. (1) We extend Object-Z with explicit guards (and implicit preconditions) to bring it closer to actions systems and hence better support the use of action refinement. (2) We take advantage of the fact that our specifications are terminating to define a version of action refinement in which there is no need to distinguish between global (observable) and local (non-observable) actions: the state is assumed to be observed initially and at termination only. (3) Most importantly, we introduce a strategy which allows a designer to iterate through 3 steps on the way to a final design. These steps build on our experience with the design process [2], [3], [8], [6].

## II. A formal notation

A specification is captured by a construct based on the class construct of Object-Z [7]. The syntax for declarations of types, constants and variables and for predicates is identical to that in Object-Z. The syntax for actions is similar to that of operations in Object-Z except that it additionally has an explicit guard.

As an example, consider the following specification of a self-configuring robot swarm [9]. The specification has a constant *target* which is a finite set of positions in 3-dimensional space representing the desired configuration of the robots. A variable *atoms* represents the positions of the finite set of robots. An invariant constrains the number of atoms to be the same as the number of positions in the target shape.

*System*1

$[Position]$

$target : \mathbb{F}\, Position$

---

$atoms : \mathbb{F}\, Position$

---

$\#atoms = \#target$

*INIT*

$true$

*Configure*

$\Delta(atoms)$

$atoms \neq target$

$atoms' = target$

### A. Semantics

The semantics of a specification is given in terms of the guarded design calculus [10]: an application of Hoare and He's Unifying Theories of Programming (UTP) [11]. A *guarded design* of the form $g \,\&\, (pre \vdash post)$ represents a reactive program that is enabled iff the guard $g$ is satisfied and, when enabled, behaves as the design $pre \vdash post$. The predicate is defined in terms of the same variables as the *design* and an additional 'status' variable *wait*. The variable *wait* denotes whether or not the control flow of the program allows the action to execute (independently of whether or not its guard is true). If $wait = true$ the program is unable to execute; otherwise the program is able to execute.

$$g \,\&\, (pre \vdash post) \;\widehat{=}\; (pre \vdash (post \wedge \neg\, wait')) \lhd g \wedge \neg\, wait \rhd (true \vdash (\vec{v}' = \vec{v} \wedge wait'))$$

where $\vec{v}$ is the vector of state variables of the program.

If a guarded design starts in a state where the guard is false, the program will set *wait* to true which prevents any subsequent programs executing. We let $g(P)$ denote the guard of a guarded design, and $D(P)$ denote the design that describes its behaviour when the guard is true.

Let $[\![.]\!]$ be a function mapping a syntactic entity to its semantics. Let *System* be a specification with constants $\vec{c}$, state variables $\vec{v}$, state invariant $inv(\vec{c},\vec{v})$, initial state predicate $init(\vec{c},\vec{v})$ and actions $Act_1,\ldots,Act_n$.

- The semantics of an action

  $Act_i$

  $\Delta(\vec{u})$

  *declarations of local variables* $\vec{x}$

  $guard(\vec{c},\vec{v},\vec{x})$

  $effect(\vec{c},\vec{v},\vec{x},\vec{v}')$

  for $1 \leq i \leq n$, is a guarded design defined as follows.

  $[\![Act_i]\!] \;\widehat{=}\; \textbf{var}\ \vec{x};\ (guard \,\&\, pre \vdash post);\ \textbf{end}\ \vec{x}$

  where

  $pre \;\equiv\; inv(\vec{c},\vec{v}) \wedge \exists\, \vec{v}' \bullet post(\vec{c},\vec{v},\vec{c}',\vec{v}')$

  $post \;\equiv\; effect \wedge \vec{c}' = \vec{c} \wedge \vec{w}' = \vec{w} \wedge inv(\vec{c},\vec{v}')$

- The semantics of the initial state schema is also a guarded design:

  $[\![INIT]\!] \;\widehat{=}\; true \,\&\, true \vdash (init(\vec{c}',\vec{v}') \wedge inv(\vec{c}',\vec{v}'))$

- The semantics of *System* is a tuple of guarded designs.

  $[\![System]\!] \;\widehat{=}\; ([\![INIT]\!],\ \neg\, wait * ([\![Act_1]\!] + \ldots + [\![Act_n]\!]))$

  where

  $b * P \;\widehat{=}\; b \wedge (P;\ b * P) \vee \neg\, b \wedge (\vec{v}' = \vec{v})$

  defines a guarded design which keeps executing $P$ while predicate $b$ is true, and

  $$P + Q \,\widehat{=}\, (g(P) \vee g(Q)) \,\&\, (g(P) \wedge D(P) \vee g(Q) \wedge D(Q))$$

  defines a guarded choice between $P$ and $Q$.

Note that we are interested only in terminating specifications (where *wait* will become true). For a non-terminating specification, the behaviour of the specification would be defined by the weakest fixed point of the iteration [11].

### B. Refinement

Let $P(\vec{v}_1,\vec{v}_1')$ and $Q(\vec{v}_2,\vec{v}_2')$ be guarded designs, and $R(\vec{v}_1,\vec{v}_2')$ be a predicate defining the retrieve relation mapping between unprimed abstract states and primed concrete states. Data refinement from $P$ to $Q$, denoted as $P(\vec{v}_1,\vec{v}_1') \sqsubseteq_R Q(\vec{v}_2,\vec{v}_2')$, can be defined as follows.

$$R(\vec{v}_1,\vec{v}_2');\ Q(\vec{v}_2,\vec{v}_2') \Rightarrow P(\vec{v}_1,\vec{v}_1');\ R(\vec{v}_1,\vec{v}_2')$$

Let $S_1$ and $S_2$ be specifications such that $[\![S_1]\!] = (I_1, P_1)$ and $[\![S_2]\!] = (I_2, P_2)$, and $R$ be a retrieve relation mapping between unprimed states of $S_1$ and primed states of $S_2$. $S_1 \sqsubseteq_R S_2$ when $(I_1 \sqsubseteq_R I_2) \wedge (P_1 \sqsubseteq_R P_2)$.

As an example, consider the following specification that refines *System*1.

*System2*

$[Position]$

$target : \mathbb{F}\, Position$

$atoms : \mathbb{F}\, Position$

$placed : \mathbb{F}\, Position$

$\#atoms = \#target$

$placed \subseteq atoms \cap target$

*INIT*

$true$

*Place*

$\Delta(atoms, placed)$

$placed \subset target$

$\exists p : target \setminus placed \bullet placed' = placed \cup \{p\}$

The refinement from *System*1 to *System*2 can be verified using the retrieve relation $R \equiv target' = target \wedge atoms' = atoms \wedge ok' = ok \wedge wait' = wait$. In this case the proof is straightforward. In general, there may be several actions in both the abstract and concrete specifications and the complexity of the specifications may make direct proof of refinement impracticable. To deal with this we detail an incremental approach to the proof of a refinement in Section III.

## III. A DESIGN STRATEGY

The goal of our design strategy is to iteratively decompose actions of our abstract system specification to concrete actions representing agent interactions. These concrete actions can then be used to inform the implementation of the agent protocols. To deal with the increase in complexity as the number of actions in both abstract and concrete specifications increases, we provide a systematic approach to deriving a correct refinement. The intuitive approach gleaned from our earlier work [2], [3], [8], [6] breaks a refinement proof into 3 steps which the designer may iterate through on the way to a final design.

### A. Simulation

The first step is to decide on a composition of concrete actions (using the sequential composition (;), guarded choice ($+$) and iteration ($*$) operators) to simulate each abstract action. This provides the basis for the concrete design. At this stage, however, we are not concerned with the enabledness of the overall system, nor the interference between simulations. Each simulation should be enabled only when the abstract action is, and should result in a post state consistent with the abstract action.

Let $S_1$ and $S_2$ be specifications and $R(\vec{v}_1, \vec{v}_2')$ be a retrieve relation mapping between unprimed states of $S_1$ and primed states of $S_2$. Let $a$ be an abstract action of $S_1$ and $C$ be a composition of the concrete actions of $S_2$ using the operators ; , $+$ and $*$. $C$ simulates $a$ iff

$$(R(\vec{v}_1, \vec{v}_2) \wedge g([\![C]\!]) \Rightarrow g([\![a]\!])) \ \wedge \ (D([\![a]\!]) \sqsubseteq_R D([\![C]\!]))$$

where $[\![C_1;\ C_2]\!] \equiv [\![C_1]\!];\ [\![C_2]\!]$, $[\![C_1 + C_2]\!] \equiv [\![C_1]\!] + [\![C_2]\!]$ and $[\![b * C]\!] \equiv b * [\![C]\!]$.

For example, the abstract action *Place* from *System*2 in Section II could be simulated by the sequential composition of two concrete actions *Recruit* and *Move*.

*Recruit*

$\Delta(atoms, waiting)$
$p, q : Position$

---

$(p, q) \in nb$
$p \in placed$
$q \in target \setminus placed$

---

$waiting' = waiting \cup \{p\}$

*Move*

$\Delta(atoms, placed)$
$p, q : Position$

---

$(p, q) \in nb$
$p \in waiting$
$q \in target \setminus placed$

---

$placed' = placed \cup \{q\}$

Let $R_1 \equiv R \wedge placed' = placed$ be a retrieve relation between the states of *System*2 and the new specification described above. The proof that *Recruit*; *Move* simulates *Place* is straightforward.

### B. Interleaving

The simulation step considers each abstract action in isolation. In our experience [2], [3], this matches the way a designer decides on the concrete actions in a design. It is necessary, however, to allow the simulations of different actions to interleave. In next step we consider the effect of interleaving on the chosen simulations and modify the concrete actions and simulations accordingly.

To do this we need to show that any sequence of actions that the concrete specification can undergo corresponds to a sequence of simulations of abstract actions. We introduce a function $\tau$ which returns the set of *action traces*, *i.e.*, sequences of actions, defined by a simulation.

Let $\Sigma$ be the set of simulations of the abstract actions. For all $C \in \Sigma$ and concrete actions $a$ the following hold.

1) If the guard of $a$ is true initially or following $C$ then $\langle a \rangle$ is an action trace of a simulation $C' \in \Sigma$.
$$(\exists \vec{v}, \vec{v}' \bullet (init(\vec{v}') \vee [\![C]\!](\vec{v}, \vec{v}')) \wedge g([\![a]\!])(\vec{v}')) \Rightarrow (\exists C' \in \Sigma \bullet \langle a \rangle \in \tau(C'))$$
2) If the guard of $a$ is true before the final action of an action trace $t \frown \langle b \rangle$ of $C$, where $t \neq \langle\,\rangle$, then $t \frown \langle a \rangle$ is an action trace of a simulation $C' \in \Sigma$.
$$\forall \langle a_1, \ldots, a_n, b \rangle \in \tau(C) \bullet (\exists v, v' \bullet ([\![a_1]\!];\ \ldots;\ [\![a_n]\!])(v, v') \wedge g([\![a]\!])(v')) \Rightarrow (\exists C' \in \Sigma \bullet \langle a_1, \ldots, a_n, a \rangle \in \tau(C'))$$

In our example, the only action enabled initially and after the simulation *Recruit*; *Move* is *Recruit*. Since it also the first action of the simulation, condition 1) holds.

During the simulation, *i.e.*, after the *Recruit* action, both *Recruit* and *Move* are enabled. While the simulation allows *Move* as the next action, it does not allow *Recruit*. Hence, condition 2) is not satisfied. To remedy this, we could strengthen the guard of *Recruit* so that only a single *Recruit* action can occur before a *Move*. This does not match our intended design, however, where multiple atoms may be recruiting neighbours at any time. The alternative is to weaken the simulation, allowing multiple *Recruit* actions before a *Move*.

The new simulation for our example is *Recruit* $\mathcal{U}$ *Move* where $P\ \mathcal{U}\ Q$ repeatedly executes guarded design $P + Q$, terminating after the first execution of $Q$. Whenever $\mathcal{U}$ is used in a simulation, the simulation will be divergent unless the terminating action can be guaranteed to occur. In our example this is not the case: *Recruit* actions could continue forever. Obviously, this means the simulation does not simulate the abstract action *Place*. To guarantee that

*Move* will occur, we strengthen the guard of *Recruit* so that a given atom only recruits once.

*Recruit*

$\Delta(atoms, waiting)$
$p, q : Position$

---

$(p,q) \in nb$
$p \in placed \setminus waiting \land q \in target \setminus placed$

---

$waiting' = waiting \cup \{p\}$

Since the number of atoms is finite, *Recruit* can only happen a finite number of times. It is then straightforward to show that *Recruit* $\mathcal{U}$ *Move* simulates *Place*.

### C. Deadlock

Once a suitable simulation is found for each abstract action, the final step is to ensure that the resulting concrete system does not deadlock (*i.e.*, reach a state where $wait = true$) more often than the abstract system.

Let $S_1$ be an abstract specification with actions $a_1, \ldots, a_n$ simulated by compositions $C_1, \ldots C_n$ of actions of a concrete specification $S_2$ under retrieve relation $R(\vec{v}_1, \vec{v}'_2)$. $S_2$ does not deadlock more than $S_1$ iff the following holds.

$$R(\vec{v}_1, \vec{v}_2) \land \neg\, (g([\![C_1]\!]) \lor \ldots \lor g([\![C_n]\!]))(\vec{v}_2) \Rightarrow \neg\, (g([\![a_1]\!]) \lor \ldots \lor g([\![a_n]\!]))(\vec{v}_1)$$

Returning to our example, the guard of *Recruit* $\mathcal{U}$ *Move* is false when both *Recruit* and *Move* are not enabled. Since $waiting \subseteq placed$ is an invariant, the guard of *Recruit* $\mathcal{U}$ *Move* is false when either

(a) there is no $p \in placed$ or,

(b) there is such a $p$, and there is no $q \in target \setminus placed$ such that $(p,q) \in nb$.

In case (b), since *target* is fully connected and $placed \subseteq target$ we can deduce that $placed = target$. Hence, $\neg\, g([\![Recruit]\!]\ \mathcal{U}\ [\![Move]\!])$ is $placed = \varnothing \lor placed = target$. However, $\neg\, g([\![Place]\!])$ is $placed = target$ and so, with $R_1$ as the retrieve relation, the deadlock condition does not hold. This tells the designer that something needs to change in the concrete specification. One possible solution is to strengthen the invariant of the concrete specification to exclude states where $placed = \varnothing$. This can be done either explicitly, or implicitly by strengthening the initial state to include $placed \neq \varnothing$ (since elements are never removed from *placed* by any action).

Effectively, we have made a design decision to start with at least one atom already fixed in a target position. The strengthened initial condition (and invariant) allows us to strengthen the retrieve relation to $R_2 \equiv R_1 \land placed' \neq \varnothing$ without invalidating the refinement between abstract and concrete initial state schemas, nor the proofs of simulation of abstract actions. The deadlock condition can then be shown to hold under $R_2$.

## IV. Conclusion

We have presented a strategy for designing self-organising systems using refinement-based action decomposition. The strategy allows us to incrementally develop a collection of agent interactions which result in a desired global state. It differs from action refinement in one significant way: changes to *global* (*i.e.*, observable) state variables can be made by introduced actions.


## Acknowledgements

The authors wish to acknowledge Jeff Sanders for his input into the ideas presented in this paper. This work was supported by Australian Research Council (ARC) Discovery Grant DP110101211.


## References


[1] G. Smith, J. W. Sanders, and K. Winter, "Reasoning about adaptivity of agents and multi-agent systems," in *17th IEEE International Conference on Engineering of Complex Computer Systems(ICECCS2012)*, 2012, pp. 341–350.

[2] G. Smith and J. Sanders, "Formal development of self-organising systems," in *6th International Conference on Autonomic and Trusted Computing (ATC 2009)*, 2009, pp. 90–104.

[3] S. Eder and G. Smith, "An approach to formal verification of free-flight separation," in *Self-Adaptive and Self-Organizing Systems Workshop (SASOW 2010)*, 2010, pp. 166–171.

[4] R. Back and J. von Wright, "Trace refinement of action systems," in *5th International Conference on Concurrency Theory (CONCUR '94)*, 1994, pp. 367–384.

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

[6] G. Smith and K. Winter, "Incremental development of multi-agent systems in Object-Z," in *Software Engineering Workshop (SEW-35)*, 2012.

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

[8] A. Sampson and G. Smith, "Gravity points in potential-field approaches to self organisation," in *Self-Adaptive and Self-Organizing Systems Workshop (SASOW 2010)*, 2010, pp. 110–115.

[9] K. Støy, "Using cellular automata and gradients to control self-reconfiguration," *Robotics and Autonomous Systems*, vol. 54, pp. 135–141, 2006.

[10] J. He, "Service refinement," *Science in China Series F: Information Sciences*, vol. 51, pp. 661–682, 2008.

[11] C. Hoare and J. He, *Unifying Theories of Programming*. Prentice Hall, 1998.