COMPUTING SCIENCE

Formal Stepwise Development of Scalable and Reliable Multiagent Systems

Denis Grotsev, Alexei Iliasov and Alexander Romanovsky

TECHNICAL REPORT SERIES

No. CS-TR-1205 **June 2010**

TECHNICAL REPORT SERIES

No. CS-TR-1205 June, 2010

Formal Stepwise Development of Scalable and Reliable Multiagent Systems

D. Grotsev, A. Iliasov and A. Romanovsky

Abstract

In this paper we consider the coordination aspect of large-scale dynamically-reconfigurable multi-agent systems in which agents cooperate to achieve a common goal. The agents reside on distributed nodes and collectively represent a distributed system capable of executing tasks that cannot be effectively executed by an individual node. The two key requirements to be met when designing such a system are scalability and reliability. Scalability ensures that a large number of agents can participate in computation without overwhelming the system management facilities and thus allow agents to join and leave the system without affecting its performance. Meeting the reliability requirement guarantees that the system has enough redundancy to transparently tolerate a number of node crashes and agent failures, and is therefore free from single points of failures. We use the Event B formal method to formally validate the design and to ensure system scalability and reliability.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

GROTSEV, D., ILIASOV A., ROMANOVSKY, A.

Formal Stepwise Development of Scalable and Reliable Multiagent Systems
[By] D. Grotsev, A Iliasov, A Romanovsky
Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2010.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1205)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-1205

Abstract

In this paper we consider the coordination aspect of large-scale dynamically-reconfigurable multi-agent systems in which agents cooperate to achieve a common goal. The agents reside on distributed nodes and collectively represent a distributed system capable of executing tasks that cannot be effectively executed by an individual node. The two key requirements to be met when designing such a system are scalability and reliability. Scalability ensures that a large number of agents can participate in computation without overwhelming the system management facilities and thus allow agents to join and leave the system without affecting its performance. Meeting the reliability requirement guarantees that the system has enough redundancy to transparently tolerate a number of node crashes and agent failures, and is therefore free from single points of failures. We use the Event B formal method to formally validate the design and to ensure system scalability and reliability.

About the author

Denis Grotsev is a 4th year PhD student at Kazakh National University, Kazakhstan.

Alexei Iliasov is a Researcher Associate at the Faculty of Computing Science of Newcastle University, Newcastle-upon-Tyne,UK. He got his PhD in Computer Science in 2008 in the area of modelling artefacts reuse in formal developments. His research interests include agent systems, formal methods for software engineering and tools and environments supporting modelling and proof .

Alexander (Sascha) Romanovsky is a Professor in the CSR. He received a M.Sc. degree in Applied Mathematics from Moscow State University and a PhD degree in Computer Science from St. Petersburg State Technical University. He was with this University from 1984 until 1996, doing research and teaching. In 1991 he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993 he was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, the University of Newcastle upon Tyne. In 1992-1998 he was involved in the Predictably Dependable Computing Systems (PDCS) ESPRIT Basic Research Action and the Design for Validation (DeVa) ESPRIT Basic Project. In 1998-2000 he worked on the Diversity in Safety Critical Software (DISCS) EPSRC/UK Project. Prof Romanovsky was a co-author of the Diversity with Off-The-Shelf Components (DOTS) EPSRC/UK Project and was involved in this project in 2001-2004. In 2000-2003 he was in the executive board of Dependable Systems of Systems (DSoS) IST Project. He has been the Coordinator of the Rigorous Open Development Environment for Complex Systems (RODIN) IST Project (2004-2007). From February 2008 he will coordinate the FP7 DEPLOY Integrated Project (2008-2012).

Suggested keywords

MOBILE AGENTS
MULTI-AGENT SYSTEMS
FORMAL SPECIFICATION
B METHOD
PROGRAM REFINEMENT
SCALABILITY
RELIABILITY
REDUNDANCY

Formal Stepwise Development of Scalable and Reliable Multiagent Systems

Denis Grotsev
Kazakh National University, Kazakhstan

Alexei Iliasov, Alexander Romanovsky
Newcastle University, United Kingdom

ABSTRACT

In this paper we consider the coordination aspect of large-scale dynamically-reconfigurable multi-agent systems in which agents cooperate to achieve a common goal. The agents reside on distributed nodes and collectively represent a distributed system capable of executing tasks that cannot be effectively executed by an individual node. The two key requirements to be met when designing such a system are scalability and reliability. Scalability ensures that a large number of agents can participate in computation without overwhelming the system management facilities and thus allow agents to join and leave the system without affecting its performance. Meeting the reliability requirement guarantees that the system has enough redundancy to transparently tolerate a number of node crashes and agent failures, and is therefore free from single points of failures. We use the Event B formal method to formally validate the design and to ensure system scalability and reliability.

INTRODUCTION

Agent systems are complex distributed systems that are dynamically composed of autonomous agents. The dynamic nature of these systems makes them applicable in situations where traditional, static architectures do not perform well. One example of such systems is a distributed system comprising a large number of low-power processing nodes, such as laptops and PDAs, that may fail, disconnect and reconnect. Assuming that the overall system purpose is to carry out some computations reliably and efficiently, the problem of managing the allocation of tasks to nodes is not trivial as one has to ensure, at the very least, that the system is capable of tolerating a certain rate of agent failures and disconnections. Due to the nature of the system, we are unable to formulate and realize constraints on the behaviour of individual system nodes. This makes it difficult to design preemptive corrective actions such as delegating a node task to another node just before the node disconnects. At the same time, there is the problem of ensuring better utilization of the processing power offered by the newly-connected nodes. Without a steady inflow of new jobs, such nodes would not improve the system performance unless there is a reconfiguration facility redistributing the existing jobs to the new nodes.

The aim of our work is to develop an approach to building reliable systems that would work over a set of unreliable distributed agents. As with any reliability solution, we will employ redundancy to improve system reliability. In particular, in our approach, we will be tasking several agents with the execution of the same job to ensure the reliability of the overall system. In the rest of the paper we refer to such a group of agents executing the same task as a bundle.

The applicability of our solution critically depends on its scalability and, in particular, on the ability to dynamically incorporate new agents into the system. Moreover, to improve the overall reliability the system should dynamically create new bundles when new agents join the system and ensure that no bundle includes more agents than necessary to tolerate a predetermined rate of agent failures/disconnections. After a number of agent disappearance events, a bundle may become so small that any further agent failure means not being able to complete the assigned task. To counter this, our

solution ensures that several bundles are merged into a single, larger and more reliable bundle before they become too small. At the same time, when a bundle becomes too large it operates less efficiently as all agents in a bundle need to be synchronized and hence the higher management overhead is required. Moreover, in this situation, the system does not benefit from the computational power of the excessive agents. To guarantee that the system does not use more resources than necessary our mechanism ensures that such a bundle is split into several smaller bundles. To achieve the scalability and reliability of such application the mechanism also dynamically balances the sizes of all its bundles.
The first scientific contribution of our work is in a formal definition of the requirements and of the main properties of the system, supporting reliable and scalable applications deployed over the network of unreliable agents. The second contribution is in a formal stepwise development of detailed models which meet these requirements. Below we show how to develop such systems formally by refinement using the Event B method.

BACKGROUND: EVENT B

Event B (Métayer, Abrial & Voisin, 2005) is a state-based formal method inherited from Classical B (Abrial, 1996). An Event B model is made of a static part, called context, and a dynamic part, called machine. A context defines constants *c*, sets *s*, axioms *P* and theorems *T*:

context *C*
sets *s*
constants *c*
axioms *P(c, s)*
theorems *T(c, s)*

A machine is described by a collection of variables *v*, invariants *I(c, s, v)*, an initialisation event *RI(c, s, v')* and a set of machine events *E*:

machine *M*
sees *C*
variables *v*
invariants *I(c, s, v)*
events *E*

In the above, construct **sees** *C* makes context *C* declarations available to machine *M*. Invariants specify safe model states and also define variable types. Events are named entities made of a guard predicate and a list of actions:

name = **any** *p* **where** *G(c, s, p, v)* **then** *R(c, s, p, v, v')*

where *p* is a vector of parameters, *G(c, s, p, v)* is a guard and *R(c, s, p, v, v')* is a list of actions. Event is enabled when guard *G* is satisfied on the current state *v*. If there are several enabled events, an enabled event is selected non-deterministically. The result of an event execution is a new model state *v'*.
The essence of the Event B method is in the verification of consistency and refinement conditions of machines. Model consistency conditions demonstrate that various parts of a machine do not contradict each other. The following is a summary of machine consistency properties.
Axioms *P* and invariants *I* should allow some values for constants, sets and variables:

$\exists\ c, s, v\ .\ P(c, s) /\backslash I(c, s, v)$

Every event, including the initialization event, must establish invariants:

$P(c, s) \land I(c, s, v) \land G(c, s, v) \land R(c, s, v, v') \Rightarrow I(c, s, v')$
$P(c, s) \land RI(c, s, v') \Rightarrow I(c, s, v')$

It should be possible to find a new state satisfying the event guard and event action conditions:

$P(c, s) \land I(c, s, v) \land G(c, s, v) \Rightarrow \exists\, v' \,.\, R(c, s, v, v')$
$P(c, s) \Rightarrow \exists\, v' \,.\, RI(c, s, v')$

The essence of the Event B development methodology is the step-wise refinement procedure where a simple model is gradually extended to include both new system phenomena and more detailed explanations of the phenomena of the abstract model.
Machine *M* can be refined by some new machine *N*. Then machine *M* is called an abstract machine in regards to machine *N*. Concrete machine *N* defines new variables *w* and must provide a gluing invariants $J(c, s, v, w)$ that links the states of *N* and *M*. A concrete event refines an abstract event by replacing the original guard with a stronger predicate $H(c, s, w)$ and defining new action $S(c, s, w, w')$. Such new action must be feasible:

$P(c, s) \land I(c, s, v) \land J(c, s, v, w) \land H(c, s, w) \Rightarrow \exists\, w' \,.\, S(c, s, w, w')$

Concrete guard *H* must strengthen abstract guard *G*:

$P(c, s) \land I(c, s, v) \land J(c, s, v, w) \land H(c, s, w) \Rightarrow G(c, s, v)$

A concrete action *S* must refine abstract action *R*:

$P(c, s) \land I(c, s, v) \land J(c, s, v, w) \land H(c, s, w) \land S(c, s, w, w') \Rightarrow$
$\exists\, v' \,.\, (R(c, s, v, v') \land J(c, s, v', w'))$

There are several other proof obligations and well-formedness rules. The complete definition can be found in (Abrial & Metayer, 2005).

SYSTEM MODEL

This section briefly introduces the properties of the system we are going to develop. In this work we design a system that serves clients by accepting, deploying and reporting the results of computational tasks. The system consists of a distributed set of agents, managed by a decentralised management unit. The aim of our work is to formally develop the design of such unit.
Agents reside on unreliable nodes. An agent may disappear (leave the system) due to a fault, node crash, network connection loss, or simply because it is willing to engage into some other activity. The exact cause does not matter provided there are means for detecting the event of an agent disappearance. In the following we assume that the agents have comparable computation power and estimate it by the least capable agent. The aim of the management unit is to distribute a task across several agents.
The task allocation procedure must be reliable in a sense that a failure of one or more agents (up to some limit) would not prevent the system from fulfilling its obligations. To accomplish this, we must ensure that at any given moment a task is allocated at least to one agent present in the system. This requirement implies some additional assumptions about the ability of an agent to leave the system. These are addressed in the model we discuss further.
The system has to scale effectively to be able to accept more tasks when more agents dynamically join the system. Consequently, a task should be sent for the execution to the least number of agents that still

allows the system to meet the reliability requirements. A well-known approach to distributing tasks uses the request hash algorithm (Ratnasamy et al, 2001) to compute the address of processing unit.
The approaches discussed in (Stoica et al, 2001; Rowstron & Druschel, 2001; Maymounkov & Mazieres, 2002; Zhao et al, 2004) propose to construct "sparse" address space where the number of address space size is fixed and is very large but only small proportion is in use at any given time. There is just one agent corresponding to a given address. When it happens that a request is routed to an unused address, the responsibility for handling the request falls onto some agent in residing in a neighbor address. Typical approach to finding neighbours is by allocating addresses using the ring or hypercube topologies.
Another approach is suggested in (Schlosser et al, 2002). The authors propose to constructs "dense" address space where the number of addresses is variable but is maintained to be equal or slightly above the overall number of agents. All the addresses are utilized and it may happen that one or few agents are associated with several addresses.
The advantage of the dense address space design over the sparse design is the possibility to reason about the workload of agent. In other words, it is possible to give the upper bound for the number of addresses an agent is associated with. In a sparse address space, this number is stochastic.
The disadvantage of the (Schlosser, 2002) design is that only one agent is responsible for an address. If this agent fails some request may be lost before the system has a chance to discover the problem. In other words, all these agents represent single points of failure.
We try to take the best from the approaches described above and define a special level in the system design that serves the purpose of reliability. In this level, agents are grouped into bundles. In our design a bundle is associated with exactly one address.
To address the problem of a single point failure caused by a crash of an agent, we group agents into bundles. Agents of a bundle are working on the same tasks and employ group communication to maintain a coherent view of a shared state space. In the development, we treat a bundle as an identifiable atomic unit characterised by the number of agents in it. A bundle that is too small threatens to undermine reliability. The strategy is to try and merge a small bundle with another bundle. If a bundle grows too large, the task duplication becomes inefficient and group communication becomes costly. The latter is to the fact that the overheads of maintaining a coherent shared state in a bundle increase quadratically with the bundle size. Thus, these larger bundles must be split into several smaller bundles.

OVERVIEW OF THE EVENT B DEVELOPMENT

The development of the system introduced in the previous section consists of eight refinement steps. Its structure is given in Figure 1. The machines of the development are shown as nodes connected by refinement arcs. An arc depicts the Event B refinement relation and goes from a concrete model to its immediate abstraction. The overall structure is a tree as in our modelling we explore several design directions. First, we concentrate on the design where agents are distributed across bundles that are merged and split independently of each other and the number of bundles linearly depends on the number of the agents.
The development starts with the modelling of the abstract notions of redundancy and efficiency in the *m0_few_many* model. The first refinement *m1_scale* introduces the abstract size of the system that determines the number of bundles and the number of tasks the system can handle. The next refinement step *m2_lower_upper* introduces the bundle size limits to detalise the abstract notions of redundancy and efficiency.
We then explore an alternative design with a tighter relation between the system bundles to achieve better reliability. This is reflected in models *m3_ready*, where we introduce the modelling of message broadcasting, *m4_width*, that models the relation between the chosen topology and a bundle size and *m5_count* where the link between agents and bundles is made explicit. Finally, to simplify the proofs, we also show how to construct a model facilitating inductive proofs.

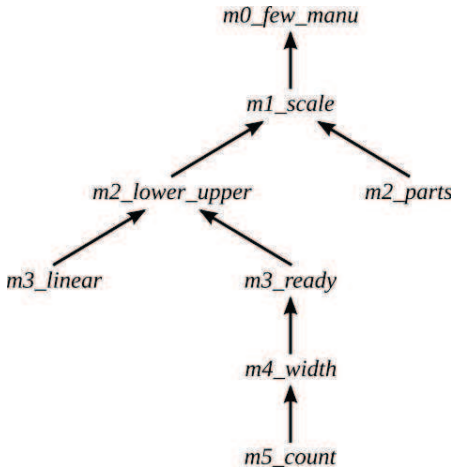

Figure 1. Development structure.

REFINEMENT STEPS OF THE DEVELOPMENT

Abstract Model

We start to specify the system at the most abstract level with two Boolean variables *few* and *many* in the *m0_few_many* machine. The first variable indicates that there are too few agents in a bundle to work reliably. The second variable is *TRUE* when there are too many agents in a bundle to work effectively. The system is in a normal state when both variables are FALSE. Otherwise the system state is considered to be exceptional and a certain reconfiguration activity should take place. Initially, the state is set to normal.

event *INITIALISATION*
then
few := *FALSE*
many := *FALSE*
end

Abstract event *underflow* signals that there are too few agents in a bundle. By doing this it switches the system state into exceptional.

event *underflow*
where
few = *FALSE*
many = *FALSE*
then
few := *TRUE*
end

The system recovers from an exceptional state by merging several bundles into a single, bigger bundle. At this stage we do not portray such details and the *merge* event corrects the exceptional state by simply flipping the *few* flag.

event *merge*
where
few = *TRUE*

then
few := *FALSE*
end

Similarly, the abstract model declares the *overflow* event to detect the formation of excessively large bundles and the *split* event simply splits such bundles into several smaller ones.
From now on we will assume that the system can handle only one exception at any given time (or, more formally, the occurrence of exceptional situations is interleaved with system reaction; note this does not set the limits on the rate at which exceptions may happen). The consequence is that the system may not have too few and too many agents simultaneously in differing bundles. This translates into a requirement of a balanced distribution of agents across bundles. Formally, the property is expressed with the following invariant:

inv1 few = FALSE ∨ *many = FALSE*

The number of agents in the system is changing gradually. In other words, agents join and leave the system independently one by one and there is no sudden increase in agent number in a given bundle. A further assumption is that the periods in which the system is in an exceptional state are not adjacent and are interleaved by periods of normal operation. There is no event for a direct transition from a state where *few=TRUE* to a state when *many=TRUE* and vice versa. This assumption allows us to avoid situations when the system engages into an endless cycle where one exception immediately follows another. The state diagram of the abstract model is given in Figure 2. Small left and large right circles show that there is a bundle containing too few or too many agents respectively. The middle circle represents the normal state.

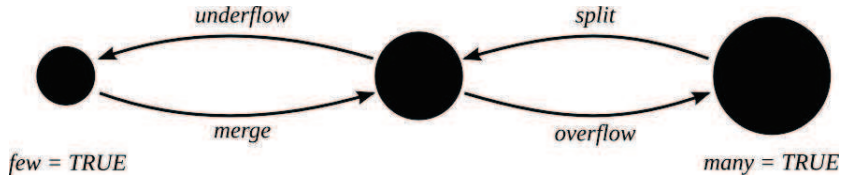

Figure 2. State diagram of the abstract model.

Scale of the System

When new agents join the system they are immediately associated with a bundle. Likewise, agents leave the system by leaving a bundle. Bundles may appear or disappear only due to some corrective actions executed by the management support. This means that there might be bundles without agents. The correction either splits too large bundles or merges bundles that are too small. In other words, the number of bundles in the system is increased by event *split* or decreased by event *merge*. New variable *scale* is introduced to give a certain (abstract) view on the number of bundles in the system. Initially, *scale* is set to zero.

inv1 scale ∈ ℕ

event *merge* **extends** *merge*
then
scale := *scale* – 1
end

Event *merge*, however violates invariant *inv1* as there is no guarantee that *scale* would not become negative. The solution is to forbid the state leading to the deadlock and strengthen the guards of the *merge* and *underflow* events.

inv2 scale = 0 => *few* = *FALSE*

event *underflow* **extends** *underflow*
where
scale > 0
end

We assume here that the management system detects the condition when a bundle is too small while it still has enough time to execute the corrective actions without losing a task. The fact that the system has more than one bundle to merge is indicated by positive values of *scale*.
A state diagram of the model is given in Figure 3. Note that it contains only the two lowest values of the infinite natural *scale* sequence. The state denoted by the grey circle is forbidden by the model invariant *inv2*.

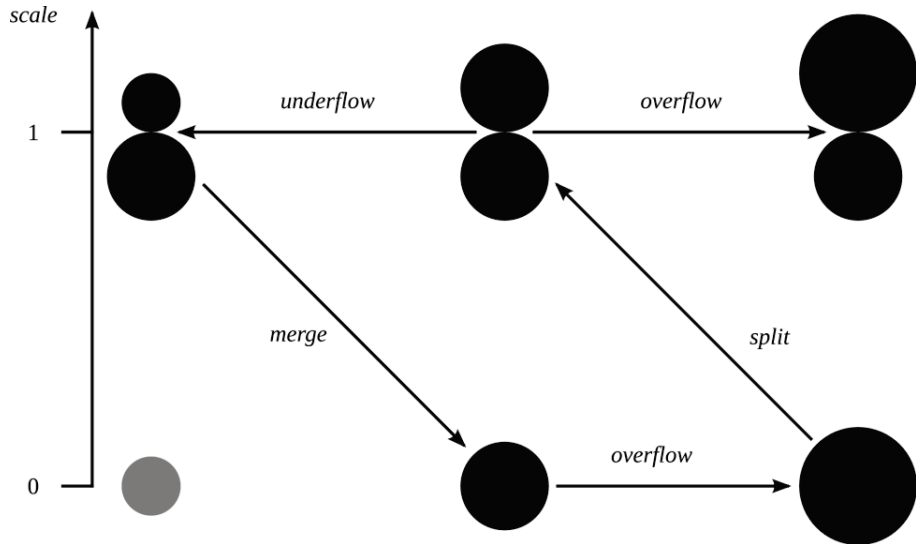

Figure 3. State diagram of the model with scale.

Limit to Bundle Size

In the previous models we abstractly refer to small and large bundles. Now we formally define these terms by refining model *m1_scale* (Figure 1) with concrete variable *scale*. Let the *LOWER* and *UPPER* constants limit the number of agents in a bundle in a normal state. If the number of agents is less then *LOWER* then a bundle is too small. If it is greater than *UPPER* it is too large.
Small bundles must still provide the level of redundancy sufficient to tolerate agents leaving the system. Therefore, the value of *LOWER* must be strictly greater than one:

axm1 LOWER > 1

In a general case, a correction of a bundle size splits a bundle into a set of bundles or merges several bundles into single one. If we assume that the correction operates on exactly two bundles then the following condition must hold:

axm2 2 * *LOWER* ≤ *UPPER*

Maintaining the information about the exact number of agents in a bundle is expensive due to node distribution. Thus, the reconfiguration logic uses a limited knowledge about the bundle states and has a somewhat imprecise picture of the overall system. Because of this the bundle size is only an estimate provided by variables *lower* and *upper* defining, correspondingly, the lower and upper bounds of the agent number in a bundle. These variables respect the following conditions:

inv1 few = *TRUE* <=> *lower* < *LOWER*
inv2 many = *TRUE* <=> *upper* > *UPPER*

Here, the *lower* estimation is always positive because the system has to be redundant and the *upper* estimation is limited because the system has to be efficiently scalable. Of course, the *lower* estimation is never greater than the *upper* estimation.

inv3 lower ≥ *LOWER* – 1
inv4 upper ≤ *UPPER* + 1
inv5 lower ≤ *upper*

Initially, *lower* is the minimal possible value satisfying the invariant:

event *INITIALISATION* **extends** *INITIALISATION*
then
lower := *LOWER*
upper :∈ *LOWER* .. *UPPER*
end

Small and large bundles are detected by the *LOWER* and *UPPER* boundaries of the normal state in the following way:

event *underflow* **extends** *underflow*
where
lower = *LOWER*
then
lower := *lower* – 1
end

Correction events *merge* and *split* update *lower* and *upper* to preserve the invariant:

event *merge* **extends** *merge*
any *l u*
where
l ≥ *LOWER*
u ≤ *UPPER*
l ≤ *u*
then
lower := *l*
upper := *u*
end

At this point we introduce new functionality to dynamically maintain an estimate of a bundle size, *lower* and *upper*, while in the normal state. It is similar to the *merge* and *split* events even though it refines the *skip* event of the abstract machine.

event *fluctuate*
any *l u*
where
few = FALSE
many = FALSE
l ∈ *LOWER .. UPPER*
u ∈ *LOWER .. UPPER*
l ≤ *u*
then
lower := *l*
upper := *u*
end

The state diagram of this model is given in Figure 4. The inner structure shows the number of agents. Constants are assigned to the lowest possible values *LOWER*=2 and *UPPER*=4. Thus, it is possible to have a bundle with two, three or four agents. A bundle with single agent is too small. And bundle with more than four agents is too large.

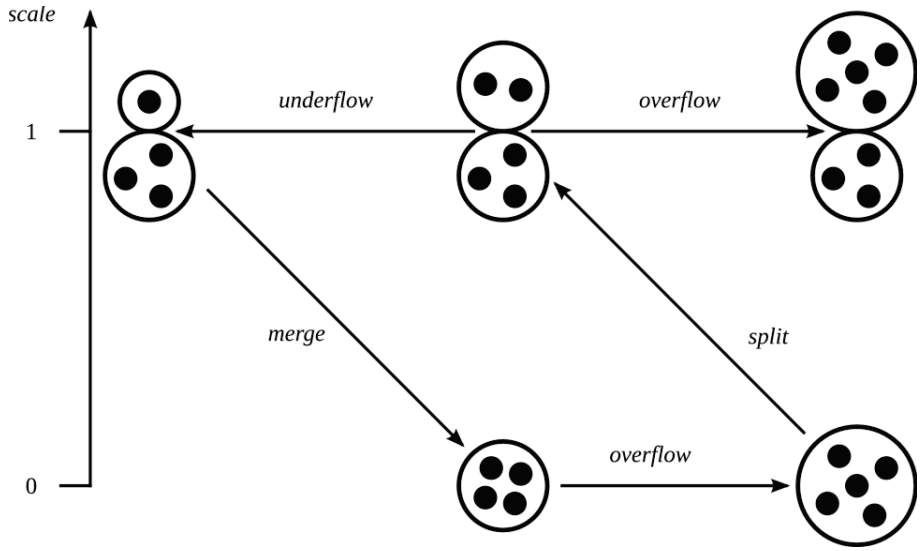

Figure 4. State diagram of the model with limited bundle size.

Linear Scale

In the previous refinement step we have ignored the distribution aspect of the system. The refined model we introduce at the next step captures the concepts that help us to make the system implementation distributed.

To do this we need to fix the granularity unit of distribution. It is difficult to transparently tolerate agent failures in a system where the distribution unit is a single agent. Hence, we introduce a bundle – a group of agents doing the same job. The agents in a bundle do not cooperate to finish a job quicker; we use multiple agents to only mask agent failures. At this point, there is no need to identify each of an agent; a

more abstract picture where the system is aware only of the number of agents in a bundle is sufficient. This brings an assumption that, at a certain level of observation, all agents are equivalent.
The *scale* of the system (the number of agents bundles) always changes by one (hence, the model name). The *split* event increases the *scale* by adding a new bundle and while *merge* removes a bundle and thus decreases the *scale*. These two events operate on a pair of bundles. This is related to the assumption that there is no broadcast communication among bundles and a pair of bundles to be merged is always found among locally-related pairs. The notion of local bundle is explained later on.
We refine the model to make the system distributed and define the *BUNDLE* set in the context. Not all bundles are present in the system at the same time and a bundle may contain several agents. We define a partial function *count* for the number of agents in a bundle in the *m3_linear* machine which refines *m2_lower_upper*.

inv1 count ∈ *BUNDLE* +–> ℕ1

In this model we assume that the value of *scale* linearly depends on the number of present bundles. This means that bundles merge and split independently of each other. Hence no broadcast or global synchronization is required in our system. The bundle number is positive because of the reliability requirement but *scale* still may be zero.

inv2 finite(*count*)
inv3 scale = card(*count*) – 1

ROOT bundle is a distinguished bundle used to bootstrap the system.

axm1 ROOT ∈ *BUNDLE*

event *INITIALISATION*
with
upper′ = *LOWER*
then
count := {*ROOT* |–> *LOWER*}
end

The *lower* and *upper* estimates of a bundle size are defined by the gluing invariant and are expressed in the terms of *count*.

inv4 ∀ *b* . *b* ∈ dom(*count*) => *lower* ≤ *count*(*b*)
inv5 ∀ *b* . *b* ∈ dom(*count*) => *upper* ≥ *count*(*b*)

A bundle with too few agents must merge with some other bundle. It is not specified here what is the other bundle. Anticipating a new refinement step, we notice the following. From the communication viewpoint, it is convenient to merge a bundle with a sibling of the bundle. To satisfy invariant *inv5*, it is necessary to ensure a certain condition on the merged bundle size. If a merged bundle happens to be exceptional (too big), the balancing event *fluctuate* would be enabled:

count := ({*b1*, *b2*} <<| *count*) U {*b1* |–> *count*(*b1*) + *count*(*b2*)}

When a bundle splits, its agents are dived equally (up to one agent, permitting odd agent count) between the new bundles. This action must re-establish invariant *inv4* (with the help of axiom *axm2*):

count := ({*b1*} <<| *count*) U {*b1* |–> *count*(*b1*)/2, *b2* |–> *count*(*b1*) – *count*(*b1*)/2}

Exponential Scale 1

A popular approach to realizing dynamically scaling systems of interconnected nodes is by constraining the way nodes are linked with the torus topology. In our case, we apply the hypercube, a special case of torus. The motivating factor is that the system diameter (the maximum number of intermediate nodes between any two nodes of a system) and, consequently, the travelling time for a message increases only as the logarithm of the total number of nodes. One notable example of a system based in this technique is a distributed hash-table algorithm underpinning many peer-to-peer network designs (Maymounkov & Mazieres, 2002). In this work nodes are sparsely allocated at the vertices of a hypercube. Furthermore, each node is associated with several vertices of hypercube by some distance function. An alternative approach (Schlosser, 2002) is to densely allocate nodes in a small-dimension hypercube.
In our case, the node of a hypercube is a bundle. When a bundle needs to split, it informs all other bundles and then every bundle of the system splits into two new bundles. This increases the topological dimension of the system by one. A similar approach is used for the case of bundle merging. When a bundle has a critically low agent number it initiates a global merge where each bundle is merged with another bundle and the total number of bundles is halved. This decreases the dimension by one. After merging or splitting, the system has to be brought back into a normal state when it is again ready for reconfiguration. At this stage, we simply denote this sate with a flag variable *ready*, though the state and the related events are introduced in detail in the following refinement steps.
We refine machine *m2_lower_upper* by *m3_ready* and add new Boolean variable showing when the system is ready for the next correction. The system is in the normal state when *few* and *many* are *FALSE* and *ready* is *TRUE*. All other states are exceptional. Initially the system is in the normal state.

event *INITIALISATION* **extends** *INITIALISATION*
then
ready := *TRUE*
end

System can detect when there are too few agents in a bundle and update the estimations of the *upper* or *lower* values.

event *underflow* **extends** *underflow*
where
ready = *TRUE*
end

The first step of a bundle merge makes the system not ready for the next correction. Events *overflow* and *split* are defined in a similar way.

event *merge* **extends** *merge*
then
ready := *FALSE*
end

The second phase of exception state correction prepares the system to the next correction by adjusting the *lower* and *upper* estimates.

event *prepare* **refines** *fluctuate*
any *l u*
where

l ∈ *lower .. upper*
u ∈ *lower .. upper*
l ≤ *u*
ready = FALSE
then
lower := *l*
upper := *u*
ready := *TRUE*
end

We assume here that the system needs to deal with only one exception at any given time.

inv1 ready = FALSE => few = FALSE
inv2 ready = FALSE => many = FALSE

State diagram of the model is given in Figure 5. The opaque figures represent intermediate states when system is not ready for the next correction and agents are involved in global communication. Constants are assigned values *LOWER*=2 and *UPPER*=8.

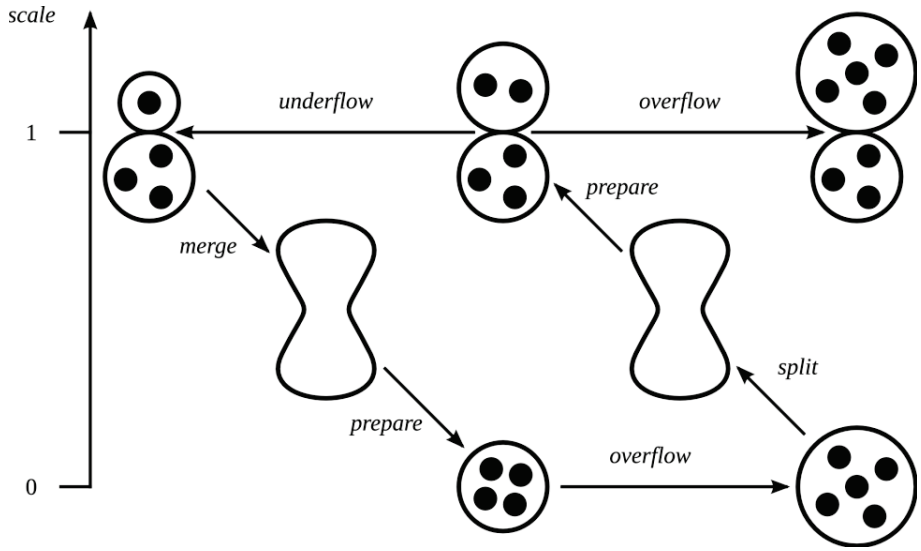

Figure 5. State diagram of the model with two phase correction.

Exponential Scale 2

When a bundle is about to send a message to initiate split or merge it would be desirable to all other bundles in a state when they can split or merge without subsequently initiating a new split or merge request. This is only possible when all the bundles contain approximately the same number of agents. To be able to reason about the comparative bundle size, we introduce new constant *WIDTH* determining the maximum difference between the sizes of any two bundles. Now, the *LOWER* and *UPPER* also take in the account the *WIDTH* value.

axm1 WIDTH > 0
axm2 2 * (*LOWER* + *WIDTH*) ≤ *UPPER*

Machine *m4_width* refines *m3_ready* to relate flag *ready*, denoting the states when the system is able to split or merge. In anormal state, the bundle size estimation is stronger to allow for the detection of the *underflow* and *overflow* conditions.

inv1 ready = TRUE => upper – lower ≤ WIDTH
inv2 few = FALSE /\ many = FALSE /\ ready = TRUE => upper – lower < WIDTH

These invariants allow us to reason about the exceptional states and to assert the theorems stating the relation between the *upper* and *lower* values. The new theorems will help us to discharge proof obligations for the *merge* and *split* events.

theorem *inv3 few = TRUE => upper < LOWER + WIDTH*
theorem *inv4 many = TRUE => lower > UPPER – WIDTH*

Merging here means that all bundles are split into pairs and each pair is consolidated into a single bundle; therefore the *merge* event effectively doubles the *lower* and *upper* estimations.

event *merge* **refines** *merge*
where
few = TRUE
then
few, *ready* := *FALSE*, *FALSE*
scale := *scale* – 1
lower, *upper* := *lower* * 2, *upper* * 2
end

Similarly, the *split* event halves them the size estimations.
The state diagram of the model is shown in Figure 6. Constants are: *LOWER*=2, *UPPER*=8 and *WIDTH*=2. A small *WIDTH* value requires more communication to archive balanced distribution of agents across bundles.

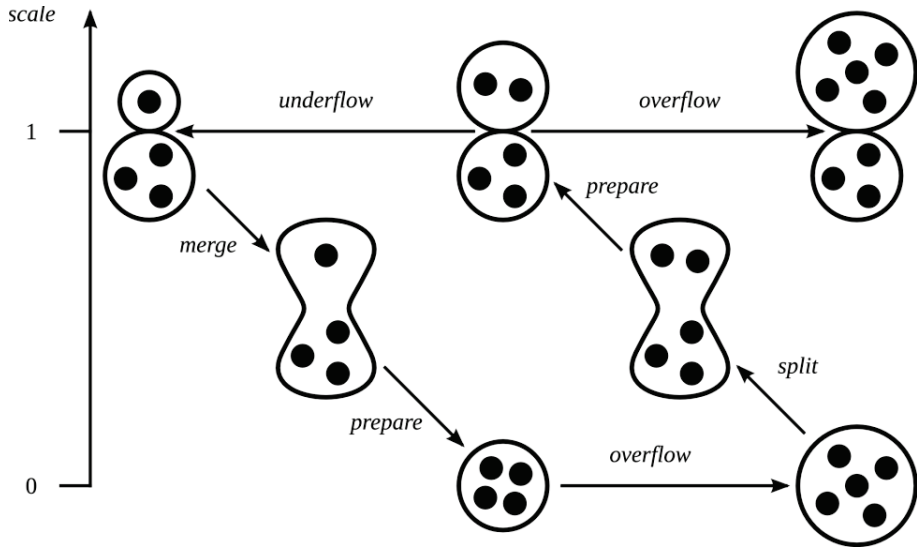

Figure 6. State diagram of the model with two phase correction limits.

Explicit Bundle Relation

The next refinement step deals with the concrete definition of a relation organizing bundles into pairs. Previously, we have assumed that bundles somehow know their neighbours and, of course the global view of neighbourhood is consistent with the local information. Realizing such a mechanism in a distributed system is far from trivial. In this model we introduce an abstract relation defining bundle pairs. Remember that new bundles appear by splitting a bundle into two. This means that, with an exception of the initial bundle, all bundles in the system have a historical parent. Importantly for us, such parent relationship defines sibling bundles – the descendants of the same parent. From the process creating new bundles, it is known that siblings come in pairs. The sibling relation gives us a ready solution for finding pairs of bundles to merge: we always merge children of the same parent. Mathematically, the relation is characterized by a binary tree with bundles as nodes. New constant *SCALE* represents the depth of the binary tree. A distinguished node *ROOT* is a tree root and the historical initial bundle. Also, initially the *scale* of the system is set to zero.

axm1 $SCALE \in BUNDLE \longrightarrow \mathbb{N}$
axm2 $SCALE(ROOT) = 0$

Every bundle has two distinct children that replace it when the bundle is split. The scale of a system containing a given child node is greater by one than the scale of the system containing the parent node.

axm3 $CHILD1 \in BUNDLE \longrightarrow BUNDLE$
axm4 $CHILD2 \in BUNDLE \longrightarrow BUNDLE$
axm5 $\forall b \cdot b \in BUNDLE \Rightarrow CHILD1(b) \neq CHILD2(b)$
axm6 $\forall b \cdot b \in BUNDLE \Rightarrow SCALE(CHILD1(b)) = SCALE(CHILD2(b))$
axm7 $\forall b \cdot b \in BUNDLE \Rightarrow SCALE(CHILD1(b)) \neq SCALE(b) + 1$
axm8 $(CHILD1 \cup CHILD2)\sim \in BUNDLE \setminus \{ROOT\} \longrightarrow BUNDLE$

In our model, we define partial function *count* to characterize the number of agents in a bundle. This description gives rise to a stronger definition of *lower* and *upper*.

inv1 $count \in BUNDLE \pfun LOWER - 1 \upto UPPER + 1$
inv2 $lower \leq \min(\mathrm{ran}(count))$
inv3 $upper \geq \max(\mathrm{ran}(count))$

Refined events *merge* and *split* use functions *CHILD1* and *CHILD2* to compute the new value *count*. Note that in the normal state the number of bundles in the system is 2^{scale}.

Recursive Ways to Model Distribution

One of the obstacles we face in the further refinement of the models is handling of the details pertaining to the scale of the system. Since the model characterizes the system for some arbitrary *scale* value (hence, it is a modelling parameter), the proofs have to be done also for the case of some arbitrary *scale*. The nature of the scaling mechanism modelling is such that the properties of a system of a given scale are naturally expressed as an extension of the properties of a system of a smaller scale. The Event B modelling language and the proof semantics do not provide means for handling complex recursive data types and, as the result, the proofs are more difficult proofs and models are less natural.

To overcome the problem, we propose to change the point of view and to define a model as a single step of a recursive process definition. In other words, we fix the scale of the system and build a model for the given scale by connecting two similar systems of smaller scales. Importantly, the definitions of model state transitions (Event B events) are the same for the main system and its sub-systems. This makes it

possible to approach the model analysis as a step of an induction procedure where *scale* is becoming the induction parameter. The induction base is a system of the zero *scale* with a single bundle.
The overall model is now a composition of two models of the previous refinement *m1_scale*. The composition process here is a simple juxtaposition of models states and events but with an addition of invariants linking the states of the composed models.

inv1 few1 = FALSE \/ many1 = FALSE
inv3 scale1 ∈ ℕ
inv5 scale1 = 0 => *few1 = FALSE*

An exception arises when any component is in exceptional state. So the *few* and *many* abstract variables are glued by disjunction of the same component variables.

inv7 few = TRUE <=> few1 = TRUE \/ few2 = TRUE
inv8 many = TRUE <=> many1 = TRUE \/ many2 = TRUE

scale is glued more complex. The *scale* of the compound machine and of its components is the same in the normal state.

inv9 few1 = FALSE /\ many1 = FALSE /\ few2 = FALSE /\ many2 = FALSE => scale1 = scale
inv10 few1 = FALSE /\ many1 = FALSE /\ few2 = FALSE /\ many2 = FALSE => scale2 = scale

But in the exceptional state one of the components lags behind the other. In this case the *scale* of the system is equal to the *scale* of the lagging component.

inv11 many1 = TRUE /\ many2 = FALSE /\ scale2 = scale1 + 1 => *scale1 = scale*
inv12 many2 = TRUE /\ many1 = FALSE /\ scale1 = scale2 + 1 => *scale2 = scale*
inv13 few1 = TRUE /\ few2 = FALSE /\ scale2 = scale1 – 1 => *scale1 = scale*
inv14 few2 = TRUE /\ few1 = FALSE /\ scale1 = scale2 – 1 => *scale2 = scale*

Note that initially both components are in the normal state.

event *INITIALISATION*
then
few1, *many1*, *scale1* := *FALSE*, *FALSE*, 0
few2, *many2*, *scale2* := *FALSE*, *FALSE*, 0
end

The *underflow1* event happens when the first component detects a too small bundle before the second component. The *merge1* event merges bundles after the second component. The *overflow1* and *split1* events are similar. The second part has the same four events.

event *underflow1* **refines** *underflow*
where
few1 = FALSE
many1 = FALSE
few2 = FALSE
many2 = FALSE
scale1 > 0
then
few1 := *TRUE*

end

event *merge1* **refines** *merge*
where
few1 = TRUE
few2 = FALSE
scale2 = scale1 – 1
then
few1 := *FALSE*
scale1 := *scale1* – 1
end

Some additional new events are omitted for brevity.

CONCLUSIONS AND FUTURE WORK

The main contribution of our work is in formal refinement of the reliability and scalability requirements together with ensuring the overall system correctness. We start from requirement specification and develop two models of the system exploring two different design approaches.

There is a large amount of research on approaches based on distributed hash tables (Stoica et al, 2001; Rowstron & Druschel, 2001; Maymounkov & Mazieres, 2002; Zhao et al, 2004) propose to distribute tasks across agents in a probabilistic way which makes it difficult to reason about reliability. We focus on explicit definition of a group of replicated agents (so called bundles).

In the system with a linear scale bundles are not tightly coupled. A bundle only has to know one other bundle to execute the merge procedure when the bundle size is too small. No message broadcasting is required and decision on splitting and merging are taken locally. Still, such system design has relies on a routing service to distribute tasks and collect results.

Designs embedding a routing service are able to address the efficiency of non-local communication explicitly. One promising direction is the hypercube topology, mainly thanks to its symmetry properties, facilitating reasoning and scalability and providing shorter communication paths (Preparata & Vuillemin, 1981; Fang et al, 2005). However, supporting a highly symmetrical topology in a dynamic environment such as ours requires additional efforts. To ensure that all bundles remain of a similar size, it is necessary to rearrange agents from larger bundles to smaller ones. In our plans is to design a balancing mechanism and introduce it as continuation refinement steps.

One critical aspect of the system is that a bundle should not be allowed to become too small for it may put in danger the fulfillment of the system obligation to its environment. To counter this, bundles that become small are merged with some other bundles. With the exponential scale design, the merging of a local pair of bundles initiates the global merging process when the total number of bundles is halved. Similarly, when a bundle has too many agents, all the bundles are split at once. Although such design handles isolated instances of small or large bundles less efficiently than the linear scale design, it scales much better to accommodate large number of bundles (and thus agents). One further reification of the design would be to consider using rational values for scale factors. This permits more accurate modeling of the process of merging of bundles of differing sizes.

To simplify the model development, we have relied on a number of assumptions about bundle states, bundle availability and the properties of the medium connecting bundles. While this is a typical approach in the modeling of a distributed system we do realize that unless the assumption are relaxed such a system may not be realised in practice (Gilbert, 2002). Our plan is to continue the development to make it closer to an implementable program.

REFERENCES

Abrial, J.-R. (2010). *Modeling In Event B: System And Software Engineering.* New York, NY, USA: Cambridge University Press.

Abrial, J.-R. (1996). *The B-book: Assigning Programs to Meanings.* New York, NY, USA: Cambridge University Press.

Colquhoun, J. & Watson, P. (2010). *A P2P Database Server Based on BitTorrent* (Tech. Rep. Series No. CS-TR-1183). Newcastle, UK: Newcastle University, School of Computing Science.

Fang, J.-F., Lee, C.-M., Yen, E.-Y., Chen, R.-X. & Feng, Y.-C. (2005). Novel broadcasting schemes on cube-connected cycles. *Communications, Computers and signal Processing, 2005. PACRIM. 2005 IEEE Pacific Rim Conference on* (pp. 629-632).

Gilbert, S. & Lynch, N. (2002) Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News, Vol. 33(2)* (pp. 51-59). New York, NY, USA: ACM.

Maymounkov, P. & Mazieres, D. (2002). Kademlia: A Peer-to-peer Information
System Based on the XOR Metric. *In Peer-to-Peer Systems, Vol.2429 of Lecture Notes in Computer Science* (pp. 53-65). Berlin: Springer.

Métayer, C., Abrial, J.R. & Voisin, L. (Ed.). (2005). *Rodin Deliverable D7: Event B language.* Project IST-511599, School of Computing Science, University of Newcastle.

Preparata, F. P. & Vuillemin, J. (1981). The cube-connected cycles: a versatile network for parallel computation. *Communications of the ACM, Vol. 24(5)* (pp. 300-309). New York, NY, USA: ACM.

Ratnasamy, S., Francis, P., Handley M., Karp R. & Shenker, S. (2001). A Scalable Content-Addressable Network. *ACM SIGCOMM 2001*. Retrieved April 3, 2010, from http://berkeley.intel-research.net/sylvia/cans.pdf.

Rowstron, A. & Druschel P. (2001). Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (pp. 329-350). Germany: Heidelberg.

Schlosser, M., Sintek, M., Decker, S. & Nejdl, W. (2002). HyperCuP — Hypercubes, Ontologies, and Efficient Search on Peer-to-Peer Networks. In Moro G. & Koubarakis M. (Ed.), *First
International Workshop on Agents and Peer-to-Peer Computing, Vol. 2530 of Lecture Notes in Computer Science* (pp. 112–124). Berlin: Springer.

Stoica, I., Morris, R.,Karger, D., Kaashoek, M. F. & Balakrishnan, H. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *Proceedings of ACM SIGCOMM '01* (pp. 149-160). San Diego, CA, USA.

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D. & Kubiatowicz, J. D. (2004). Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications: Special Issue on Recent Advances in Service Overlay Network. Vol. 22(1)* (pp. 41-53).

Keyword: Mobile agents, multi-agent systems, formal specification, B Method, program refinement, scalability, reliability, redundancy.