Dependability and Computer Engineering:

Concepts for Software-Intensive Systems

Luigia Petre
Åbo Akademi University, Finland

Kaisa Sere
Åbo Akademi University, Finland

Elena Troubitsyna
Åbo Akademi University, Finland

ENGINEERING
SCIENCE REFERENCE

Senior Editorial Director: Kristin Klinger
Director of Book Publications: Julia Mosemann
Editorial Director: Lindsay Johnston
Acquisitions Editor: Erika Carter
Development Editor: Michael Killian
Production Editor: Sean Woznicki
Typesetters: Keith Glazewski, Natalie Pronio, Jennifer Romanchak
Print Coordinator: Jamie Snavely
Cover Design: Nick Newcomer

Published in the United States of America by
Engineering Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Dependability and computer engineering: concepts for software-intensive systems / Luigia Petre, Kaisa Sere and Elena Troubitsyna, editors.
p. cm.
Summary: “This book offers a state-of-the-art overview of the dependability research, from engineering various software-intensive systems to validating existing IT-frameworks and solving generic and particular problems related to the dependable use of IT in our society”--Provided by publisher.
Includes bibliographical references and index.
ISBN 978-1-60960-747-0 (hardcover) -- ISBN 978-1-60960-748-7 (ebook) -- ISBN 978-1-60960-749-4 (print & perpetual access) 1. Reliability (Engineering) 2. Computer systems--Reliability. 3. Computer engineering. I. Petre, Luigia, 1974- editor. II. Sere, K. (Kaisa), 1954- editor. III. Troubitsyna, Elena, 1970-, editor.
TS173.D47 2011
620’.00452--dc22

2011011401

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 4
Formal Stepwise Development of Scalable and Reliable Multiagent Systems

Denis Grotsev
Kazakh National University, Kazakhstan

Alexei Iliasov
Newcastle University, UK

Alexander Romanovsky
Newcastle University, UK

ABSTRACT

This chapter considers 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 allows 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. The Event B formal method is used to validate the design formally and to ensure system scalability and reliability.

INTRODUCTION

The variety and ubiquity of modern computational devices raise the problem (and create the opportunity) of utilizing and orchestrating their processing capabilities within an integral approach which would ensure that the system using them is scalable and reliable. In our work we refer to such computational resources as system nodes. Our solution is based on the universal principle of dealing with complexity by introducing a particular level of abstraction that allows us to

DOI: 10.4018/978-1-60960-747-0.ch004

focus on achieving certain system properties. In particular, our aim is to demonstrate how properties of solutions can be formally reasoned about at various levels of abstraction.

Examples of such abstraction levels that allow developers to support integration of many nodes can be found in peer networks (BitTorrent), cloud platforms (Google App Engine) and distributed file systems. Such systems are designed to achieve required system properties. The most important one is scalability, which ensures a linear or almost linear increase in system performance with the increase in the number of nodes. Another critical system property is reliability, which allows clients to see the system as if it was realised on a single fault-free node. Due to the nature of these systems, node failures are not uncommon and should not normally lead to an overall failure or require explicit actions at the level of applications deployed on the system. In other words, within certain limits, node failures should be masked. This is typically achieved through node and application redundancy, whereby the same activity is executed on several nodes. Crucially, in case of node failures the system is automatically reconfigured.

Our work proposes a formal step-wise development model which allows us to prove the scalability and reliability of the solutions using the Event-B method. As part of our rigorous system development, we demonstrate how to formally specify a reconfiguration of the system topology performed as a response to a change in the number of nodes. We apply a multiagent approach in which a special programming unit, an agent, resides on every node and reacts to node failures and system changes in such a way as to automatically reconfigure the system to an acceptable state.

BACKGROUND: EVENT-B

Event-B (Abrial, 2010) is a state-based formal method inherited from Classical B (Abrial, 1996). It is an approach for realising industrial-scale developments of highly dependable software. The method has been successfully used in the development of several real-life applications. An Event-B development starts from creating a formal system specification. The basic idea underlying stepwise development in Event-B is to design the system implementation gradually, by a number of correctness preserving steps called refinements.

The unit of a development is a model. An Event-B model is made of the static part, called a context, and the dynamic part, called a machine. A context defines constants *c*, sets (user defined types) *s*, and declares their properties in 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*. The model invariants specify safe model states and also define variable types. An event is a named entity made of a guard predicate and a list of actions and has the following syntax:

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 for execution 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. The machine consistency conditions demonstrate that various parts of a machine do not contradict each other. The following is a summary of these conditions.

Axioms P and invariants I should be satisfiable for some values for constants, sets and variables:

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

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

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

$$P(c, s) \wedge RI(c, s, v') \Rightarrow I(c, s, v')$$

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

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

$$P(c, s) \Rightarrow \exists\ v'\ .\ RI(c, s, v')$$

The main development methodology of Event-B is refinement - the process of transforming an abstract specification while preserving its correctness and gradually introducing implementation details. Let us assume that the refinement machine N is a result of refinement of the abstract machine M. Then machine M is called an abstract machine in regards to machine N.

machine N
refines M
sees C_1
variables w
invariants $J(c_1, s_1, v, w)$
events E_1

Concrete machine N defines new variables w and provides a gluing invariants $J(c, s, v, w)$ that links the states of N and M. A concrete event from E_1 refines an abstract event by replacing the original guard $G(c, s, v)$ with a stronger predicate $H(c, s, w)$ and defining new action $S(c, s, w, w')$. Such new action must be feasible:

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

Concrete guard H must strengthen abstract guard G:

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

A concrete action S must refine abstract action R:

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

$$\exists\ v'\ .\ (R(c, s, v, v') \wedge J(c, s, v', w'))$$

The refined model can also introduce new events. In this case, we have show that these new events are refinements of implicit empty (*skip*) events of the abstract model. There are several other proof obligations and well-formedness rules. The complete definition can be found in (Abrial & Metayer, 2005).

SYSTEM MODEL

Due to their reliance on message passing rather than common shared memory, distributed computing environments pose a number of challenges, including, for example, how to make decisions about resource location and ownership, and how

to define patterns which structure communication between system nodes. In this work we mainly focus on the latter area, while only partially addressing the rest.

We are developing a system which will include the distribution of computation tasks among its nodes as its core function. By the node we understand a fairly independent computing platform – one that is free to disappear, or fail, or decline to execute a task. Since we are aiming to produce a system with many thousands of nodes, we use an abstraction of a node called *bundle*. A bundle is a collection of nodes with an additional property, which is that the agents of a bundle are able to communicate more efficiently among themselves than with those of other bundles. We do not discuss in this work how a communication infrastructure supporting bundles may be designed and deployed.

A node, i.e. a member of bundle, is associated with one or more agents (run/located on that node), whose purpose is to execute system tasks. An agent can be responsible for managing several tasks; equally, the same task may be replicated to several agents. The system maintains the following relation between its tasks and agents:

$$distribution \in Task \leftrightarrow Agent$$

In a general case, several agents can reside on the same node. The distinction between nodes and agents allows us some flexibility at the abstraction level, as it is possible to choose to see a specific execution unit as a collection of nodes or as a single node with several agents. To a great extent, the distinction between the two views is determined by how much of a unit is likely to fail or disappear from the system. If the likely scenario is an isolated failure of a unit part, it is convenient to treat the unit is a collection of independent nodes. On the other hand, if it is known that the unit is likely to fail as a whole, it is convenient to view it as a single node with multiple agents. In rough terms, the number of the agents in a node correlates with the processing capabilities of the node.

We do not distinguish here between catastrophic hardware failures of nodes, communication problems, the decision of a node to leave the system and many other failure scenarios, as we believe it is difficult in practice to have a mechanism that distinguishes between these scenarios in a system that is intrinsically open (which means, for example, that its nodes may appear and disappear at any time).

Initially, the abstract view of the system assumes that only one agent resides on a node and that agents join and leave the system independently of one another. In other words, we treat an agent as a synonym for a node and define a bijection $Agent \rightarrowtail\!\!\!\!\!\!\rightarrow Node$. The task distribution relation above can be replaced by

$$distribution \in Task \leftrightarrow Node$$

To achieve reliability, a task is replicated on several nodes:

$$\forall\, task \,.\, \mathrm{card}(distribution[\{task\}]) > 1$$

As the distribution relation given above is hard to maintain in a distributed system, a design decision was made to restrict it. We assume that agents are assigned to whole bundles rather than individual agents. In this case, all the agents of a bundle have the same set of tasks assigned to them:

$$distribution \in Task \leftrightarrow\!\!\!\!\!\!+ \; Bundle$$

We will now show how to perform certain bundle and task operations on a distribution relation of this kind. Every bundle is required to contain at least two agents. This allows us to use the bundle as an abstraction that hides node failures, treating bundles as perfectly reliable entities. We will also show how to form such bundles in an open environment. At this level of abstraction agents are interchangeable, and a bundle is characterized by its size, i.e. the number of agents in the bundle. A newly joining agent appears in one

of the existing bundles and immediately becomes engaged in the current task of its parent bundle.

We implement the distribution function by splitting it into two parts: static and dynamic.

The static part of the function, which does not change during the execution of the system, is known to every agent of the system. We are using the hash function value of a data identifier, which in our case is intended for load balancing among bundles, as such static part. Maintaining the static function does not require extra communication between system nodes.

In order to support effective search and change operations, the dynamic part must be simple and symmetric (it should not have a single point of failure) for all bundles. The B tree structure (Bayer, & McCreight, 1972) is an example of data structure with such properties. It efficiently handles changes but is not sufficiently symmetric for our purposes, since it relies on a single root that would be a single point of failure in our system. We employ a different node topology, which enjoys similar change properties but is more symmetric, as there is no global root and any bundle can be used as a root for a search tree.

The topology we use is a hypercube, in which vertices represent bundles and every edge connects exactly two bundles. A bundle plays a similar role to that of a block in a B tree. A new agent joining a bundle can lead to the latter becoming too large (compared to the average bundle size). This bundle will then need to be split into two smaller ones, each inheriting an equal share of agents as well as tasks of the original one. Conversely, agents leaving bundles will result in some of them becoming too small; it will be necessary to merge such a bundle with another one and bring together their task sets.

In terms of system topology, the merging and splitting of bundles lead to the joining and splitting of the hypercube topology vertices, which can violate its levels of symmetry and result in a topology different from that of a hypercube. For a number of reasons, it is essential to rely on the symmetry properties of a hypercube; hence, bundle operations must always result in creation of a hypercube of a differing dimension rather than in simply adding or removing nodes. Hence, when a bundle is split, all bundles need to be split, and the dimension of the hypercube increases by one. When any pair of bundles is merged all the adjacent bundle pairs are also merged, and the dimension decreases by one. In other words, at the top level of abstraction, the system reacts to a change in the number of available agents by changing its topology, which always remains a hypercube topology, and a topology change results in a hypercube of a greater or lesser dimension.

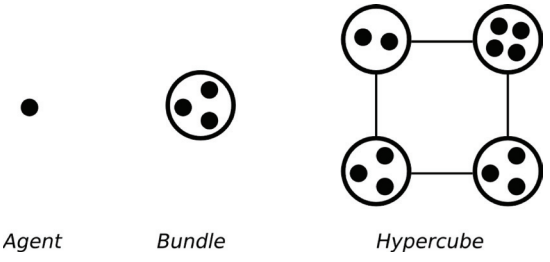
REFINEMENT STEPS OF THE DEVELOPMENT

Initial Model

The departure point for modelling is the definition of the requirements for bundles. At the most abstract level, the state of the system is represented by two Boolean variables: *few* and *many*. In the normal state, which is also the initial state, both variables are set to false.

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

Figure 1. Abstraction levels: unreliable agents, reliable bundles, scalable hypercube



When one of the bundles of the system becomes too small, event *underflow* switches the system state from the normal into an exceptional one, by raising flag *few*. Similarly, when one of the bundles becomes too large, event *overflow* raises flag *many*.

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

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

The system recovers from these exceptional states in events *merge* and *split*, respectively, by simply resetting the flags.

event *merge*
where
few = *TRUE*
then
few := *FALSE*
end

event *split*
where
many = *TRUE*
then
many := *FALSE*
end

At this point we assume that the system has to handle only one failure at a time (or, more formally, the occurrence of exceptional situations is interleaved with system reaction; note that this does not impose any limitations on the rate at which exceptions may happen). One important consequence of this is that the system may not have too few and too many agents simultaneously in differing bundles. This translates into a requirement for a balanced distribution of agents across bundles. Formally, the property is expressed with the following invariant:

inv1 few = *FALSE* ∨ *many* = *FALSE*

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.

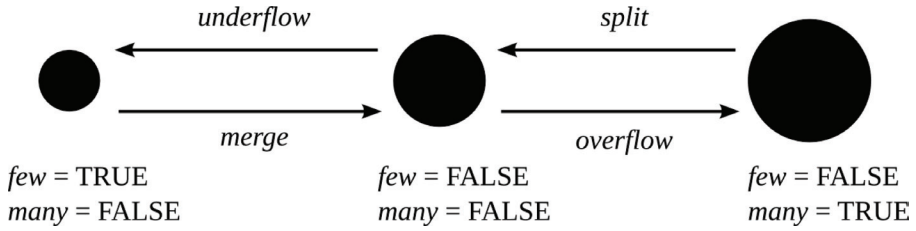
Scale of the System

The next modelling step, the first refinement in the Event-B terms, gives a certain (abstract) view of how the hypercube transforms. The new variable *scale* is introduced to denote the hypercube dimension. The total count of bundles in the system is 2^{scale}, as every edge of the hypercube consists of exactly two bundles. Initially, *scale* is set to zero and the hypercube is just a single bundle. Below, the keyword **extends** means that all declarations of the event *INITIALIZATION* of the previous abstract machine are implicitly copied into the event *INITIALIZATION* of the current concrete machine. In this case these declarations consist of two assignments of variables *few* and *many*.

inv1 scale ∈ N

event *INITIALISATION* **extends** *INITIALIZATION*
then
scale := 0
end

Figure 2. State diagram of the abstract model



Event *split*, in addition to its abstract actions, increases the dimension of the hypercube, while the event *merge* decreases it.

event *split* **extends** *split*
then
scale := *scale* + 1
end

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

The event *merge* can, however, violate the invariant *inv1*, as there is no guarantee that *scale* will not become negative. This can be resolved by introducing a new invariant to forbid the state leading to a violation and to strengthen the guards of the *merge* and *underflow* events.

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

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

A state diagram of the model is given in Figure 3. Note that it illustrates only the situations when *scale* = 0 and *scale* = 1. The circles in this figure denote bundles. The two adjacent circles mean that the 1-dimentional hypercube (segment) has only two bundles. The state denoted by the grey circle is forbidden by the model invariant *inv2*.

Modelling Bundle Size Constraints

At the abstract level of the initial model, we refer to small and large bundles. Now we formally define these terms by further refining the model above, which introduced the variable *scale*. Let the *LOWER* and *UPPER* constants define the number of agents in a bundle in a normal state. If the number of agents is below *LOWER*, then the bundle is too small. If it is above the *UPPER*, it is too large.

Small bundles must still provide a level of redundancy sufficient to tolerate agents leaving the system. Therefore, the value of *LOWER* must be higher than one:

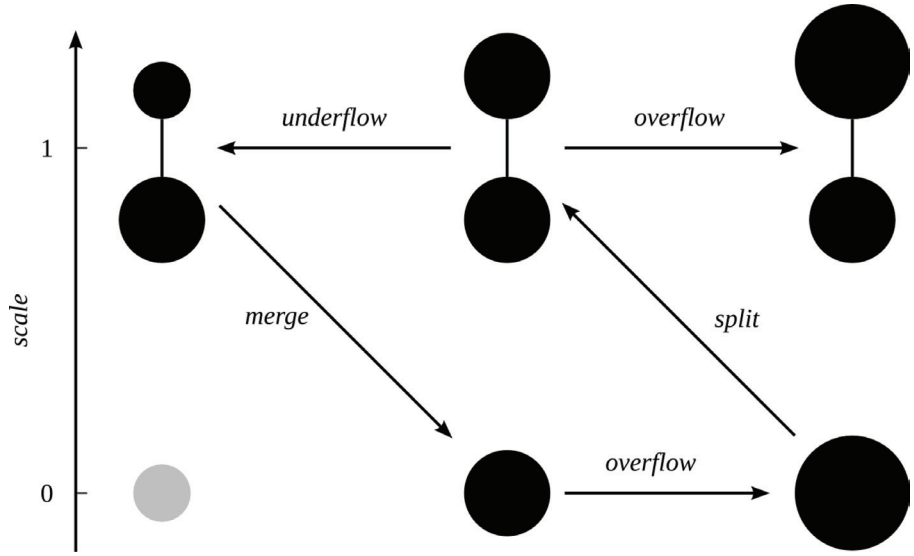
axm1 LOWER > 1

A modification of the bundle size can result in a bundle splitting into two or a pair of bundles merging into one. Therefore, the following condition must hold:

axm2 2 * *LOWER* ≤ *UPPER*

Because of node distribution, maintaining the information about the exact number of agents in a bundle is expensive. Thus, the reconfiguration logic has a limited knowledge of bundle states and has to work with an imprecise view of the overall system state. This is why the bundle size

Figure 3. State diagram of the model with scale



is only known as an estimate of the range between the variables *lower* and *upper*, which define the minimum and maximum of the agent number of all bundles, respectively. These variables observe 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, while the *upper* estimation is limited because the system has to be efficiently scalable. The *lower* estimation must never exceed the *upper* estimation.

inv3 lower $\geq$ *LOWER* – 1

inv4 upper $\leq$ *UPPER* + 1

inv5 lower $\leq$ *upper*

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

event *INITIALISATION* **extends** *INITIALISATION*
then
lower:= *LOWER*
upper:$\in$ *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

event *overflow* **extends** *overflow*
where
upper = *UPPER*
then
upper:= *upper* + 1
end

Bundle size correction events *merge* and *split* update *lower* and *upper* to preserve the invariant:

event *merge* **extends** *merge*
any *l u*
where
$l \geq LOWER$
$u \leq UPPER$
$l \leq u$
then
lower:= *l*
upper:= *u*
end

At this point of development we are ready to introduce new functionality to dynamically maintain an estimate of a bundle size, *lower* and *upper*, while in the normal state.

event *fluctuate*
any *l u*
where
few = FALSE
many = FALSE
$l \in LOWER .. UPPER$
$u \in LOWER .. UPPER$
$l \leq u$
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. For example, constants are assigned with the lowest possible values *LOWER*=2 and *UPPER*=4 according to axioms *axm1* and *axm2*. Thus, it is possible to have a bundle with two, three or four agents. A bundle with a single agent is too small and must be merged. A bundle with more than four agents is too large and is to be split.

Prepare to Correction

In the next refinement step we improve the model by explaining the notion of a normal state as a combination of two new states. The purpose is to be able to reason about the readiness of the system to perform reconfiguration.

When a system is ready to reconfigure the state is marked by flag *ready*. Initially the flag is on.

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

The system is able to detect when there are too few agents in a bundle and update the estimations of the *upper* or *lower* values while it is ready.

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

Bundle merge makes the system unready for the next correction.

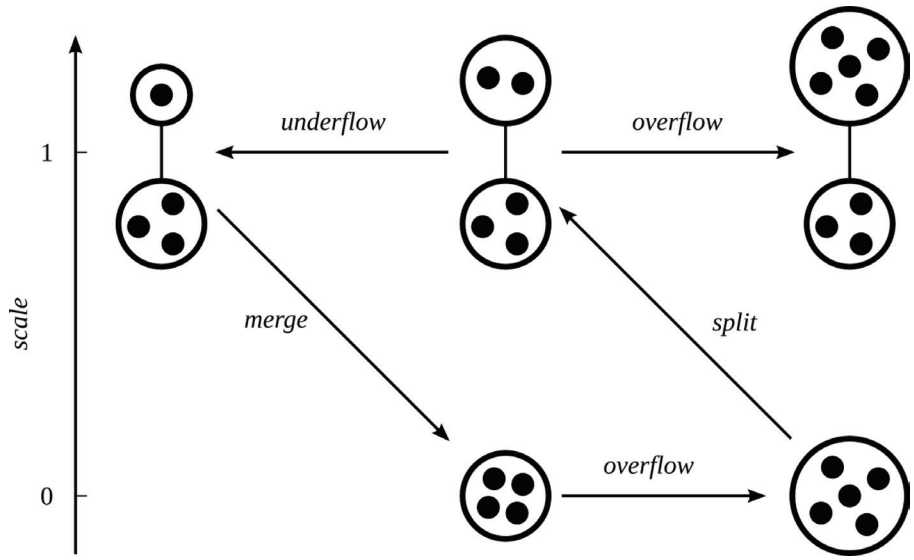
event *merge* **extends** *merge*
then
ready := *FALSE*
end

Events *overflow* and *split* are defined in a similar way.

New behaviour describes how the system prepares to the next reconfiguration by adjusting the *lower* and *upper* estimates.

event *prepare* **refines** *fluctuate*
any *l u*
where
$l \in lower .. upper$
$u \in lower .. upper$
$l \leq u$

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



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

At this stage an assumption is made that exceptions may occur only when the system is in the *ready* state.

inv1 ready = FALSE => few = FALSE

inv2 ready = FALSE => many = FALSE

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

Concerted Preparation

When a bundle is about to initiate split or merge it sends a message to all other bundles. It would be desirable that they were in a state in which 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, *LOWER* and *UPPER* also take in the account the *WIDTH* value.

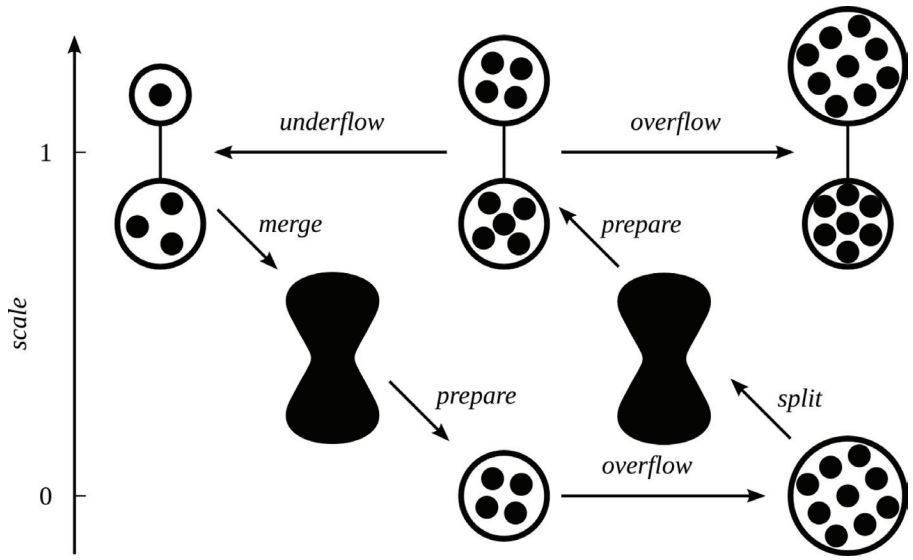
axm1 WIDTH > 0

axm2 2 * (*LOWER* + *WIDTH*) ≤ *UPPER*

At this step flag *ready* is refined. In an abnormal state, the bundle size estimation is stronger to allow for the detection of the *underflow* and *overflow* conditions (see Box 1).

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 (see Box 2).

Figure 5. State diagram of a model realising two-stage correction



Box 1.

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

Box 2.

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

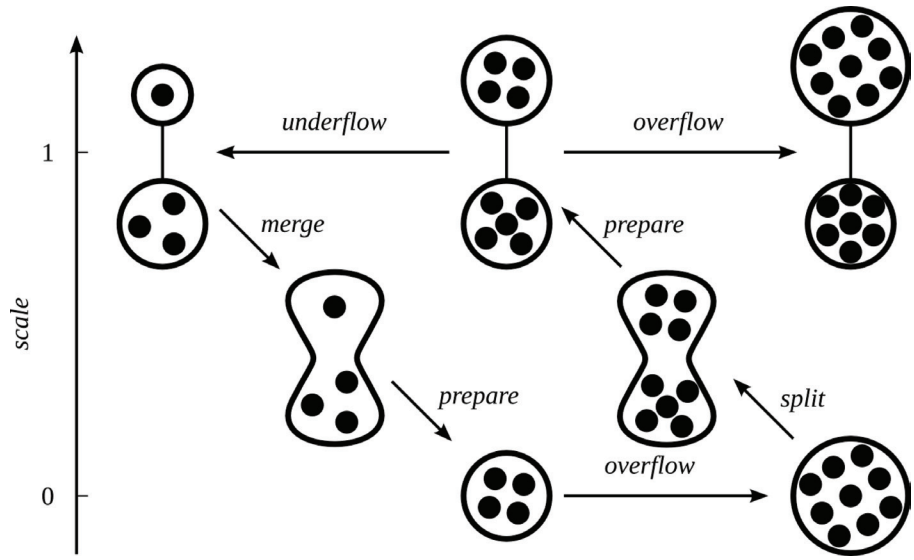
Similarly, the *split* event halves the size estimations.

The state diagram of the model is shown in Figure 6. The constants are defined as follows: *LOWER*=2, *UPPER*=8 and *WIDTH*=2. A small value of *WIDTH* leads to more communication to ensure a balanced distribution of agents across bundles.

Modelling Bundle Relations

The next refinement step introduces a relation that organizes bundles into pairs. Previously, we have assumed that bundles somehow know their neighbours and, moreover, 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

Figure 6. State diagram of a model realising concerted two-stage correction



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. It is important for us that such a parent relationship defines sibling bundles – the descendants from the same parent. The described process for creating new bundles ensures that siblings always come in pairs. The sibling relation gives us a ready solution for finding pairs of bundles to merge: we always merge two children bundles of the same parent and gain this parent bundle. Mathematically the relation is characterized by a binary tree with bundles represented as its nodes. New constant *SCALE* represents the depth of the node in the binary tree. The distinguished node *ROOT* is a tree root with zero depth and the initial bundle.

axm1 SCALE $\in$ *BUNDLE* —> 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. The parent for any bundle, except *ROOT,* may be found by inversing one of the child functions (see Box 3).

In our model, we define a 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* +–> *LOWER* – 1 .. *UPPER* + 1

inv2 lower $\leq$ min(ran(*count*))

inv3 upper $\geq$ max(ran(*count*))

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

Recursive Specification for Model Distribution

One of the obstacles we face in the further refinement of our models is handling the details

Box 3.

axm3 CHILD1 ∈ BUNDLE —> BUNDLE
axm4 CHILD2 ∈ BUNDLE —> BUNDLE
axm5 ∀ b . b ∈ BUNDLE => CHILD1(b) ≠ CHILD2(b)
axm6 ∀ b . b ∈ BUNDLE => SCALE(CHILD1(b)) = SCALE(CHILD2(b))
axm7 ∀ b . b ∈ BUNDLE => SCALE(CHILD1(b)) ≠ SCALE(b) + 1
axm8 (CHILD1 U CHILD2)~ ∈ BUNDLE \ {ROOT} —> BUNDLE

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. In our case the hypercube consists of two hypercubes of the previous dimension. 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 sometimes more difficult to prove 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 (the Event-B events) are the same for the main system and for its sub-systems. This makes it possible to approach the model analysis described below 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 first refinement which introduced variable *scale*. The composition process Box 4 is a simple juxtaposition of model states and events but with an addition of invariants linking the states of the composed models.

An exception arises when any component is in the exceptional state. So the abstract variables *few* and *many* are glued by a disjunction of the same component variables (see Box 5).

A gluing invariant for the variables *scale* is more complex. The *scale* of the compound machine and of its components are the same in the normal state (see Box 6).

According to invariant *inv8* the exceptional state of the whole system may be caused by the exceptional state of the first component while the second may be already be in a normal state. While the scale of the first component is still less than that of the second one (the first component reconfiguration lags behind the second one), invariant *inv11* defines the *scale* of the system to be equal to the *scale* of the smaller component (see Box 7). Other three invariants define the scale of the whole system in similar cases.

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

Box 4.

inv1 few1 = FALSE \/ many1 = FALSE
inv2 few2 = FALSE \/ many2 = FALSE
inv3 scale1 ∈ N
inv4 scale2 ∈ N
inv5 scale1 = 0 => few1 = FALSE
inv6 scale2 = 0 => few2 = FALSE

Box 5.

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

Box 6.

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

Box 7.

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

second component. The *merge1* event merges the bundles. The *overflow1* and *split1* events are similar. The second component 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

Other new events are omitted due to space limitations.

RELATED WORKS

The dynamic function that distributes tasks across agents (see the System Model section above) is realised by implementing a topology connecting agents and routing tasks. Two levels of abstraction will be introduced here.

The first one groups agents into bundles to ensure reliability. If the state of a bundle is changed, its agents have to communicate to quickly move to a consistent state. A bundle also needs to be partition-tolerant (Gilbert, 2002). Therefore, a fully connected topology is optimal for connecting agents in a bundle. This tends to be expensive, but because of the limited bundle size in our case, the cost of a fully connected topology is acceptable.

The second one connects bundles to ensure an efficient routing of tasks. Here efficiency means that bundles are directly connected to few neighbours, and that the distance between any pair of bundles is small.

We distinguish two kinds of topologies supporting effective routing: regular and irregular. The regular topology can be analytically expressed in design time, while the irregular topology can only be constrained in design time and reified only in run time of the system. Obviously, regular topologies are easier to verify formally.

One promising direction is the hypercube (Schlosser, 2002) and cube-connected cycles (Preparata & Vuillemin, 1981) topologies, mainly thanks to their symmetry properties, which facilitate reasoning and scalability and provide shorter communication paths (Fang et al, 2005). However, supporting a highly symmetrical topology in a dynamic environment such as ours requires additional effort, including dealing with reliability considerations. The disadvantage of the hypercube design (Schlosser, 2002) is that in this case only one agent is responsible for a hypercube vertex. If this agent fails, some requests may be lost before the system has a chance to discover the problem. In other words, all these agents represent single points of failure.

There is a large amount of research on irregular topologies that aims to distribute tasks across agents in a probabilistic way (Stoica et al, 2001; Rowstron & Druschel, 2001; Maymounkov & Mazieres, 2002; Zhao et al, 2004). Because it is based on distributed hash tables, it does not allow explicit reasoning about ensuring reliability through fault tolerance or about introducing the required level of redundancy. It is worth noticing, however, that applying hash functions (Ratnasamy et al, 2001) to routing tasks could be useful for load balancing of bundles.

FUTURE WORK

Our work can be expanded in several directions.

One of them is allowing several agents to be located on the same physical node. This will mean that in the worst case all agents of a bundle reside in the same single node. If the node fails, all the bundle tasks are lost despite agent redundancy. To resolve this problem, the bundle agents will need to be automatically placed into different physical nodes. A similar situation arises in ensuring coarse-grained reliability, when agents of a bundle are placed on distant server racks or even data centres.

Another direction is ensuring smoother scaling. In our case exactly two bundles may be merged or split. Therefore, the count of bundles in the system is a power of 2. A further reification of our design would be to consider using rational values lower than 2 for scale factors. In this case, for instance, two bundles will be split into three and three bundles into four. The system will have 2, 3, 4, 6, 8, 12, 16 and so on bundles, which will allow us to reduce the task traffic between bundles while scaling.

A third direction is to design a balancing mechanism and introduce it as the next refinement step. In the current design we specify it very briefly in the events *fluctuate* and *prepare*. This mechanism needs to ensure that all bundles preserve a similar size. This can be achieved by moving agents from larger bundles to smaller ones.

The fourth, most challenging direction is to generalize our formal approach to support irregular topologies. Such topologies are promising because they require fewer resources to maintain a high symmetry. Therefore, in theory irregular topologies will be more efficient.

CONCLUSION

The main contribution of our work is the formal development, by refinement, of large-scale dynamically reconfigurable multi-agent systems. This development meets the reliability and scalability requirements while ensuring the overall system correctness. In this development groups of replicated agents (so-called bundles) are explicitly defined, starting at a certain abstraction level, for reasoning about reliability. One critical aspect of the system is that a bundle cannot be allowed to

become too small, as the system may not be able to fulfil its obligations to its environment in that case. To tackle this, bundles that become small are merged with others. The merging of a local pair of bundles initiates a global merging process that halves the bundle total. Similarly, when a bundle has too many agents, all the bundles are split at once. Such design scales well to accommodate large numbers of bundles (and thus agents). In fact, this design is an implementation of a hypercube topology to connect bundles and route tasks.

We started our formal development from requirement specifications, developing two models of the system to explore two different design approaches. One of them relies on as a straightforward solution to how bundles are to be merged and split. Another one recursively decomposes the entire system into two components with an interface similar to that of the whole. To simplify the model development, we rely on a number of assumptions about bundle states, bundle availability and the properties of the medium connecting bundles. While this is a typical approach to modelling a distributed system, we do realize that unless these assumptions are relaxed, it may not be possible to realise such a system in practice (Gilbert, 2002). Our plan is to continue the development to bring it closer to an implementable program.

ACKNOWLEDGMENT

A. Iliasov and A. Romanovsky are supported by the FP6 ICT DEPLOY Integrated Project and by the EPSRC/UK TrAmS Platform Grant.

REFERENCES

Abrial, J.-R. (1996). *The B-book: Assigning programs to meanings*. New York, NY: Cambridge University Press. doi:10.1017/CBO9780511624162

Abrial, J.-R. (2010). *Modeling in event-B: System and software engineering*. New York, NY: Cambridge University Press.

Abrial, J.-R., & Metayer, L. V. (Eds.). (2005). *Rodin deliverable D7: Event B language. (Rodin project - IST-511599)*. UK: School of Computing Science, Newcastle University.

Bayer, R., & McCreight, E. (1972). Organization and maintenance of large ordered indexes. [Berlin, Germany: Springer.]. *Acta Informatica*, *1*(3), 173–189. doi:10.1007/BF00288683

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*. 2005 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (pp. 629-632).

Gilbert, S., & Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services. [New York, NY: ACM.]. *ACM SIGACT News*, *33*(2), 51–59. doi:10.1145/564585.564601

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

Preparata, F. P., & Vuillemin, J. (1981). The cube-connected cycles: A versatile network for parallel computation. [New York, NY: ACM.]. *Communications of the ACM*, *24*(5), 300–309. doi:10.1145/358645.358660

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). Heidelberg, Germany: Springer.

Schlosser, M., Sintek, M., Decker, S., & Nejdl, W. (2002). HyperCuP—Hypercubes, ontologies, and efficient search on peer-to-peer networks. In G. Moro & M. Koubarakis (Ed.), *First International Workshop on Agents and Peer-to-Peer Computing, Vol. 2530 of Lecture Notes in Computer Science* (pp. 112–124). Berlin, Germany: 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*, *22*(1), 41–53.

KEY TERMS AND DEFINITIONS

B Method: A tool-supported formal method based around the Abstract Machine Notation, used in the development of computer software.

Formal Specification: Is a mathematical description of software or hardware that may be used to develop an implementation.

Mobile Agents: A mobile agent is a composition of computer software and data which is able to migrate from one location to another autonomously and continue its execution on the destination location.

Multi-Agent Systems: A multi-agent system is a system composed of multiple interacting agents.

Program Refinement: The verifiable transformation of an abstract (high-level) formal specification into a concrete (low-level) executable program. Stepwise refinement allows this process to be done in stages.

Redundancy: The provision of multiple interchangeable components to perform a single function in order to cope with failures and errors.

Reliability: The ability of a system or a component to perform its required functions under stated conditions for a specified period of time.

Scalability: A desirable property of a system which indicates its ability to either handle growing amounts of work in a graceful manner or to be enlarged.