Formal Derivation of Distributed MapReduce

Inna Pereverzeva[1,2], Michael Butler[3], Asieh Salehi Fathabadi[3],
Linas Laibinis[1], and Elena Troubitsyna[1]

[1] Åbo Akademi University, Turku, Finland
[2] Turku Centre for Computer Science, Turku, Finland
[3] University of Southampton, UK
{inna.pereverzeva, elena.troubitsyna, linas.laibinis}@abo.fi
{mjb, asf08r}@ecs.soton.ac.uk

Abstract. MapReduce is a powerful distributed data processing model that is currently adopted in a wide range of domains to efficiently handle large volumes of data, i.e., cope with the big data surge. In this paper, we propose an approach to formal derivation of the MapReduce framework. Our approach relies on stepwise refinement in Event-B and, in particular, the *event refinement structure* approach – a diagrammatic notation facilitating formal development. Our approach allows us to derive the system architecture in a systematic and well-structured way. The main principle of MapReduce is to parallelise processing of data by first mapping them to multiple processing nodes and then merging the results. To facilitate this, we formally define interdependencies between the map and reduce stages of MapReduce. This formalisation allows us to propose an alternative architectural solution that weakens blocking between the stages and, as a result, achieves a higher degree of parallelisation of MapReduce computations.

Keywords: formal modelling, Event-B, refinement, event refinement structure, MapReduce

1 Introduction

MapReduce is a widely used framework for handling large volumes of data [5]. It allows the users to automatically parallelise computations and execute them on large clusters of computers. Essentially, the computation is performed in two stages – map and reduce. The first stage maps the input data to multiple processing nodes, while the second stage performs parallel computations to merge the obtained results. Typically, execution of the map stage is blocking, i.e., execution of the reduce stage does not start until the map stage is completed. Though MapReduce is already a highly performant framework, to keep pace with the drastically increasing volume of data, it would be desirable to loosen the coupling between the stages and hence exploit the potential for parallelisation to the fullest.

In this paper, we undertake a formal study of the MapReduce framework. We formally model the control flow and data interdependencies between the map and

reduce tasks, as well as derive the conditions under which the execution of the reduce stage can overlap with the execution of the map stage. Our formalisation of the (generic) MapReduce framework relies on the Event-B method and the associated Rodin platform. Event-B [1] is a formal approach that is particularly suitable for the development of distributed systems. The system development in Event-B starts from an abstract specification that is transformed into a detailed specification in a number of correctness-preserving refinement steps. In this paper, the Event Refinement Structure approach [3, 6] is used to facilitate the refinement process. The technique provides us with an explicit graphical representation of the relationships between the events at different levels of abstraction and helps to gradually derive the complex MapReduce architecture.

Event-B relies on proof-based verification that is integrated into the development process. The Rodin platform [10] automates development in Event-B by generating the required proof obligations and automatically discharging a part of them. Via abstraction, proof and decomposition, Event-B enables reasoning about system-level properties of complex distributed systems. In particular, it allows us to explicitly define interdependencies between the processed data and derive the conditions under which an execution of the reduce stage can start before completion of the map stage. We believe that the proposed approach provides the designers with a formally grounded insight on the properties of MapReduce and enables fine-tuning of the framework to achieve a higher degree of parallelisation.

The rest of the paper is organised as follows. In Section 2 we describe the generic MapReduce framework and our formalisation of it. In Section 3 we give an overview of the Event-B formalism and the Event Refinement Structure (ERS) approach. In Section 4 we present our formal derivation of the MapReduce framework in Event-B using the ERS approach. As a result, we derive two alternative architectures of the MapReduce framework – blocking and partially blocking. In Section 5 we overview the related work and present some concluding remarks.

2 MapReduce

2.1 Overview of MapReduce

MapReduce is a programming model for processing large data sets. It has been originally proposed by Google [5]. The framework is designed to orchestrate the work on distributed nodes, run various computational tasks in parallel, providing at the same time for redundancy and fault tolerance. Distributed and parallelised computations are the key mechanisms that make the MapReduce framework very attractive to use in a wide range of application areas: data mining, bioinformatics, business intelligence, etc. Nowadays it is becoming increasingly popular in cloud computing. There exist different implementations of MapReduce, among them open-source Hadoop [2], Hive [11], and others.

The MapReduce computational model was inspired by the *map* and *reduce* functions widely used in functional programming. A MapReduce computation is composed of two main steps: the *map stage* and the *reduce stage*. During the map stage, the system inputs are divided into smaller computational tasks,

which are then performed in parallel (provided there are enough processors in the cluster). The obtained collective results then become the inputs for the reduce stage, which combines them in some way to produce the overall output. Once again, the reduce inputs are split into smaller computational tasks that can be executed in parallel.

The MapReduce framework can be tuned to perform different data transformations by the user-supplied map and reduce functions. These functions encode basic mapping and reduction tasks to be performed in single nodes. The MapReduce framework then incorporates the provided functions and orchestrates the overall distributed computations based on them.

A typical example illustrating MapReduce computations is counting the word occurrences in a large set of documents. The input data set is split into smaller portions and the user-provided map function is applied to each such data block. The *map* function simply assigns to each word it encounters the value equal to 1. Overall, the map stage produces a collection of (word,1) pairs as intermediate results. Then, during the reduce stage, the user-supplied reduce function takes a portion of these intermediate data related to a particular word and sums all the occurrences of that word. Such a computation is done for each encountered word. The overall result is a set of (word,number) pairs.

2.2 Towards Formal Reasoning about MapReduce

In this section, we present a formalisation of the MapReduce framework. Specifically, we mathematically represent all MapReduce execution stages, i.e., the required data and control flow, and identify the computational (map and reduce) tasks that can be executed in parallel. Moreover, we formally define possible data interdependencies between the map and reduce tasks. The latter allows us to propose an alternative architectural solution, which weakens blocking between the MapReduce phases and, as a result, achieves a higher degree of parallelisation of MapReduce computations. In Section 4, we will propose two alternative formal developments of the MapReduce framework in Event-B, both of which rely on the formalisation presented below.

Let *IData* be an abstract type defining the input data to be processed within the MapReduce framework and *OData* be an abstract type defining the resulting output data. In a nutshell, a MapReduce computation processes the given input data and generates some result. Thus, it can be formally represented as a function:

$$MapReduce \in IData \rightarrow OData.$$

More specifically, a MapReduce computation can be defined as a functional composition of the following phases: *MSplit*, *Map*, *RSplit*, *Reduce*, and *Combine*:

$$MapReduce = MSplit;\ Map;\ RSplit;\ Reduce;\ Combine.$$

Let us note that the phases *MSplit* and *Map* together correspond to the *map stage* mentioned in Section 2.1, while the phases *RSplit* and *Reduce* belong to the *reduce stage*.

The MapReduce process starts with the *MSplit* phase. During this phase, the input data are split into a number of blocks (portions of the input data),

which can be handled independently of each other. In the following *Map* phase, the user-provided *map* function is applied to each such input block. Next, in the *RSplit* phase, the MapReduce framework groups together all the intermediate results obtained after the *Map* phase to prepare for the reduce computations. Similarly to the *MSplit* phase, the data are divided into blocks that can be handled separately. After that, the *Reduce* phase is executed, during which the user-supplied *reduce* function is repeatedly applied (once per each block). Finally, in the *Combine* phase, all the obtained results are combined into the final output.

Formalisation of the MapReduce execution phases. Next we define all the MapReduce execution phases in more detail. In the *MSplit* phase, the input data are split into a number of blocks that are later supplied to the *map* function. To emphasise the independent nature of map computations, we associate the notion of a *map task* with such a portion of the input data to be processed separately.

Let *MTask* be a set of all possible map tasks and *MData* be an abstract type defining the data obtained after the splitting. Then the *MSplit* phase can be mathematically represented as follows:

$$MSplit \in IData \rightarrow (MTask \nrightarrow MData).$$

Essentially, *MSplit* produces a partitioning of the input data to be used in the *Map* phase among different map tasks. Note that the result of *MSplit* is a partial function since only a subset of *MTask* may be needed for particular input data.

We assume that the input data fully determines the number and the subset of involved map tasks.[4] To extract this information, we use the following functions

$$mtasks \in IData \rightarrow \mathbb{P}_1(MTask), \quad mnum \in IData \rightarrow \mathbb{N}_1$$

defined as

$$\forall idata \in IData \cdot\ mtasks(idata) \ = \ \mathsf{dom}(MSplit(idata)),$$
$$\forall idata \in IData \cdot\ mnum(idata) \ = \ \mathsf{card}(MSplit(idata)),$$

where dom and card are the function domain and set cardinality operators.

The *Map* phase involves transformation of all the data obtained by the *MSplit* phase into the intermediate form to be used in the later phases. Let *RData* be an abstract type defining the intermediate data obtained after the *Map* phase. Then *Map* phase can be mathematically represented as the following function:

$$Map \in (MTask \nrightarrow MData) \rightarrow \mathbb{P}_1(MTask \times RData).$$

Therefore, *Map* takes the map data partitioning produced by *MSplit* and returns the transformed data associated with the map tasks that produced them. These results then become the input data for the following reduce computations.

In our formalisation the *Map* results consist of a set of (*mtask, rdata*) pairs, without assuming any further structure among them. This is done intentionally, since grouping and partitioning of these data will be performed in the *RSplit* phase.

[4] This applies only to the involved computational tasks. Actual software components that will be employed to carry out the necessary computations can be dynamically assigned and re-assigned for a specific map or reduce task.

All the involved map tasks should be performed within the *Map* phase. Formally, this requirement can be formulated as follows:

$$\forall f \in MTask \nrightarrow MData \cdot f \neq \varnothing \Rightarrow \mathsf{dom}(f) = \mathsf{dom}(Map(f)).$$

Next the results obtained by the *Map* phase are grouped together to prepare for reduce computations. Similarly to the *MSplit* phase, they should be first partitioned among the individual *reduce tasks*.

Let *RTask* be a set of all possible reduce tasks. Then the *RSplit* phase can be formally defined as the following function:

$$RSplit \in \mathbb{P}_1(MTask \times RData) \rightarrow (RTask \nrightarrow \mathbb{P}_1(RData)).$$

Essentially, the function takes the intermediate results produced by the *Map* phase and produces data partitioning among the involved reduce tasks.

We can reason about the actual number and the subset of the involved reduce tasks. Once again, this is determined by the original input data. Formally, we introduce the functions

$$rtasks \in IData \rightarrow \mathbb{P}_1(RTask), \quad rnum \in IData \rightarrow \mathbb{N}_1$$

defined as

$$\forall idata \in IData \cdot rtasks(idata) = \mathsf{dom}(RSplit(Map(MSplit(idata)))),$$
$$\forall idata \in IData \cdot rnum(idata) = \mathsf{card}(RSplit(Map(MSplit(idata)))).$$

The *RSplit* phase only rearranges the intermediate data, producing their partitioning among the reduce tasks. Therefore, neither new data should appear nor any of the existing data can disappear during this transformation. Mathematically, this can be formulated as the following property:

$$\forall f \in \mathbb{P}_1(MTask \times RData) \cdot \mathsf{ran}(f) = (\bigcup rt \in \mathsf{dom}(RSplit(f)) \mid RSplit(f)(rt)),$$

where ran is the function range operator.

The *Reduce* phase is similar to the *Map* phase – it takes as input a data partitioning produced by *RSplit* and returns transformed data:

$$Reduce \in (RTask \nrightarrow \mathbb{P}_1(RData)) \rightarrow \mathbb{P}_1(OData),$$

where *OData* is an abstract type defining the resulting output data.

Finally, the last *Combine* phase can be simply defined as follows:

$$Combine \in \mathbb{P}_1(OData) \rightarrow OData.$$

Formalisation of the `map` and `reduce` functions. The *Map* phase is based on repeated invocations of the user-supplied function `map`. The `map` function can be formally represented in the following way:

$$\texttt{map} \in MData \rightarrow \mathbb{P}_1(RData).$$

Thus, it takes an input data from *MData* and produces some intermediate data to be used in reduce computations. The `map` function and the *Map* phase are tightly linked. To be precise, the union of all the results obtained from all the `map` function applications should be equal to the overall result of the *Map* phase:

$$Map = \{f \cdot f \in MTask \nrightarrow MData \mid f \mapsto (\bigcup mt \cdot mt \in dom(f) \mid \{mt\} \times map(f(mt)))\}.$$

The user-supplied `reduce` function can be specified as follows:

$$\texttt{reduce} \in \mathbb{P}_1(RData) \rightarrow \mathbb{P}_1(OData).$$

It takes as an input a subset of the reduce data *RData* and produces some subset of output data from *OData*.

Finally, the overall result of the *Reduce* phase should be equal to the combined results obtained by repeated application of the `reduce` function:

$$Reduce = \{f \cdot f \in RTask \nrightarrow \mathbb{P}_1(RData) \mid f \mapsto (\bigcup rt \cdot rt \in dom(f) \mid \texttt{reduce}(f(rt)))\}.$$

Essentially, the *Reduce* definition is directly based on the user-supplied `reduce` function.

Formalisation of interdependencies between the map and reduce tasks. The main principle of MapReduce is that all the map and reduce computations are distributed to multiple independent processing nodes. The reduce inputs are based on the previously produced map outputs. However, in some cases, the reduce inputs might depend on only particular map outputs. Therefore, the reduce stage can be initiated before all the map computations are finished. To relax the limitation of the original MapReduce computation flow, requiring that the reduce stage starts only after completing the map stage, we formally define the *dependence relation* between the map and reduce tasks as the following function *dep*:

$$dep \in IData \rightarrow \mathbb{P}(RTask \times MTask),$$

with the following property:

$$\begin{aligned} \forall\, idata \in IData,\, rt \in RTask,\, mt \in MTask \cdot \quad & rt \mapsto mt \in dep(idata) \quad \Leftrightarrow \\ & mt \in \mathsf{dom}(MSplit(idata)) \wedge \\ & (\exists rd \in RData \cdot rt \in \mathsf{dom}(RSplit(Map(MSplit(idata))))) \ \wedge \\ & rd \in RSplit(Map(MSplit(idata)))(rt) \ \wedge \ mt \mapsto rd \in Map(MSplit(idata))). \end{aligned}$$

The property states that for any input data *input*, a map task *mt* and a reduce task *rt* are in *dependence relation* (i.e., a reduce task depends on a map task), if and only if some intermediate data *rd* has been generated for this reduce task *rt* by the computations of the map task *mt* during the *Map* phase. Essentially, the relation *dep* defines the data interdependencies between the map and reduce stages. This formalisation allows us to propose (in Section 4) an alternative architectural solution that weakens blocking between the stages.

Finally, to make it possible for a particular reduce task to start immediately after all the necessary data have been produced by the map tasks related by *dep*, we need a version of *RSplit*, defining a partial split related with a specific reduce task. For a given reduce task, it produces the grouped together results obtained within the Map phase:

$$\texttt{rsplit} \ \in \ RTask \nrightarrow (\mathbb{P}_1(MTask \times RData) \rightarrow \mathbb{P}_1(RData)).$$

Again, the union of the results obtained from all the `rsplit` function applications should be the result of the *RSplit* phase:

$$\begin{aligned} & \forall f \cdot f \in \mathbb{P}_1(MTask \times RData) \Rightarrow \\ & \qquad RSplit(f) = (\bigcup rt \cdot rt \in dom(rsplit) | \{rt \mapsto \texttt{rsplit}(rt)(f)\}). \end{aligned}$$

In Section 4 we will demonstrate that, by relying on the proposed formalisation, we can derive a formal model of the MapReduce framework. There we will propose two models of MapReduce – *blocking* and *partially blocking* models.

3 Formal Development by Refinement: Background

3.1 Event-B

Event-B is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving [1]. In Event-B, a system model is specified using the notion of an *abstract state machine*. An abstract state machine encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the dynamic behaviour of a modelled system. The variables are strongly typed by the constraining predicates that, together with other important system properties, are defined as model *invariants*. Usually, a machine has an accompanying component, called a *context*, which includes user-defined sets, constants and their properties given as a list of model axioms.

The dynamic behaviour of the system is defined by a collection of atomic *events*. Generally, an event has the following form:

$$e \;\widehat{=}\; \textbf{any}\ a\ \textbf{where}\ G_e\ \textbf{then}\ R_e\ \textbf{end},$$

where e is the event's name, a is the list of local variables, and (the event *guard*) G_e is a predicate over the model state. The body of an event is defined by a *multiple* (possibly nondeterministic) assignment to the system variables. In Event-B, this assignment is semantically defined as the next-state relation R_e. The event guard defines the conditions under which the event is *enabled*, i.e., its body can be executed. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B employs a top-down refinement-based approach to system development. A development starts from an abstract specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps, we gradually reduce nondeterminism and introduce detailed design decisions. The consistency of Event-B models, i.e., verification of model well-formedness, invariant preservation as well as correctness of refinement steps, is demonstrated by discharging the relevant proof obligations. The Rodin platform [10] provides an automated support for modelling and verification. In particular, it automatically generates the required proof obligations and attempts to discharge them.

3.2 Event Refinement Structure

The Event Refinement Structure (ERS) [3, 6] approach augments Event-B refinement with a graphical notation that allows us to explicitly represent the relationships between the events at different abstraction levels as well as define the required event sequence in a model. ERS is illustrated by example in Figure 1. The diagram explicitly shows that *AbstractEvent* is refined by *Event2*,

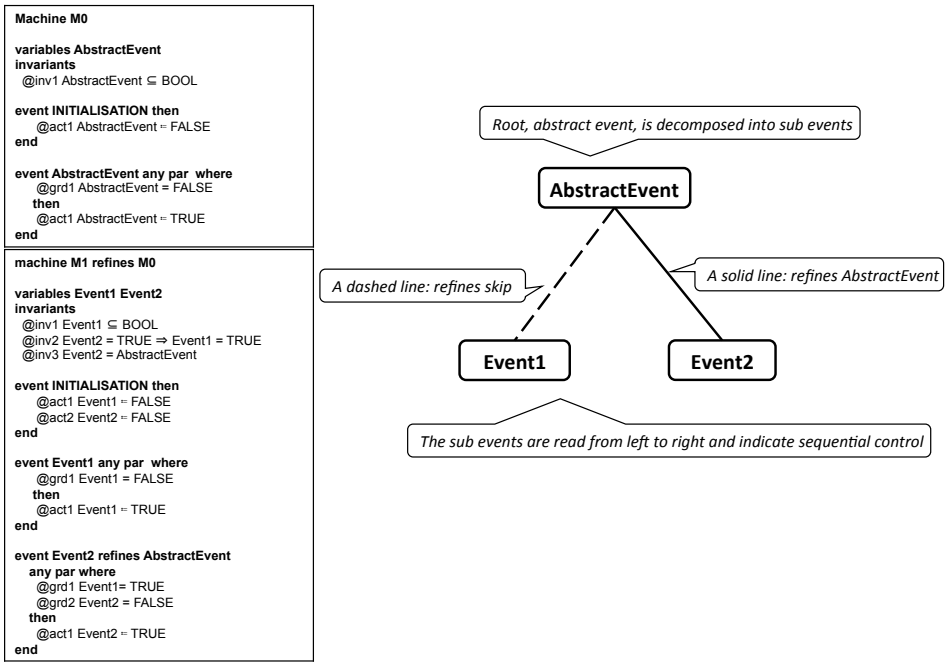

Fig. 1. Event Refinement Structure (ERS) Diagram

while *Event1* is a new event that refines *skip*. Moreover, the diagram shows that the effect achieved by *AbstractEvent* in the abstract machine is realised in the refining machine by the occurrence of *Event1* followed by *Event2*.

In ERS, the sequential execution of the leaf events is depicted from left to right. The event sequencing is managed by additional control variables introduced into the underlying Event-B model. For instance, for each leaf event (node) represented in Fig. 1, there is one boolean control variable with the same name as the event. When the event Event1 occurs, the corresponding control variable is set to *TRUE*. The following event, *Event2*, can occur only after *Event1*. This is achieved by checking the value of the *Event1* control variable in the guard of *Event2*.

Boolean variables only allow controlling single execution of events. When multiple executions of an event are needed, the event is parameterised and set control variables are used instead of boolean ones. This allows the event to occur many times with different values of its parameter. A parameter can be introduced in an event by the ERS constructors. The ERS constructors used in this paper are illustrated by two simple examples in Fig. 2. The use of *all* constructor indicates that *Event1* is executed for all instances of the p parameter before execution of *Event2*, while the use of the constructor *some* indicates that *Event1* is executed for some of instances of the p parameter before execution of *Event2*. The corresponding control variables for *Event1* and *Event2* are defined as sets in the model.

Fig. 2. ERS *all* /*some* Constructors

Event-B adopts an event-based modelling style that facilitates the correct-by-construction development of complex distributed systems. Since MapReduce is a framework designed for large-scale distributed computations, Event-B is a natural choice for its formal modelling and verification.

4 Formal Development with Event Refinement Structure

In this section, we rely on our formalisation presented in Section 2.2 to develop two alternative Event-B models of the MapReduce framework: *blocking* and *partially blocking*. The presented formal developments make use of the Event Refinement Structure (ERS) approach, presented in Section 3.2. Our development strategy is based on gradually unfolding all the MapReduce computational phases by refinement. Such small model transformation steps allow us to efficiently handle the complexity of the MapReduce framework.

Let us note that our development of the MapReduce framework is generic. It relies on the use of abstract functions to represent essential data transformations of MapReduce. These abstract functions can be treated as generic system parameters that can be later instantiated with their concrete instances for the specific MapReduce implementations.

4.1 Blocking Model of MapReduce

The mathematical data structures and their properties from our MapReduce formalisation constitute the basis for defining the Event-B *context* component that is used throughout the whole formal development. Essentially, the whole presented formalisation is incorporated as the context, e.g.

$$\texttt{axm8:}\ MSplit \in IData \rightarrow (MTask \nrightarrow MData)$$
$$\texttt{axm9:}\ Map \in (MTask \nrightarrow MData) \rightarrow \mathbb{P}_1(MTask \times RData)$$
$$\texttt{axm10:}\ RSplit \in \mathbb{P}_1(MTask \times RData) \rightarrow (RTask \nrightarrow \mathbb{P}_1(RData)),\ ...$$

We will constantly rely on these definitions to ensure the correctness of the overall data transformation process within our MapReduce models. Since the formalised definitions are still abstract (generic), our presented development essentially formally describes a family of possible MapReduce implementations. Due to space limit, we do not present the complete development but rather give its graphical representation using the ERS graphical notation. The full Event-B models of this development can be found in [8].

Abstract model of MapReduce. We start with an abstract model in which the whole MapReduce computation is done in one atomic step. This behaviour is modelled by the event OutputMapReduce:

OutputMapReduce $\widehat{=}$
any $t1,\ t2,\ t3,\ t4$
when $t1 = MSplit(idata) \wedge t2 = Map(t1) \wedge t3 = RSplit(t2) \wedge t4 = Reduce(t3)$
then $output := Combine(t4)$ **end**

With help of the ERS approach, we decompose the atomicity of OutputMapReduce into smaller steps. Verification of the refinement proof obligations ensures that the decomposition preserves correctness. Specifically, in the next several consecutive refinement steps, we break the atomicity of the OutputMapReduce event by introducing explicit events for the following MapReduce phases: *MSplit*, *Map*, *RSplit*, and *Reduce*. Fig. 3 presents the ERS diagram of the model.

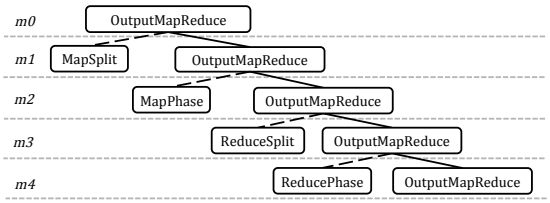

Fig. 3. Blocking model: ERS diagram (for OutputMapReduce)

The new model events MapSplit, MapPhase, ReduceSplit and ReducePhase specify the sequential execution of the MapReduce phases. The sequence between the events is enforced by following the rules given in Section 3.2. It is also specified by the invariant properties on the control variables:

$$OutputMapReduce = TRUE \Rightarrow ReducePhase = TRUE,$$
$$ReducePhase = TRUE \Rightarrow ReduceSplit = TRUE,$$
$$ReduceSplit = TRUE \Rightarrow MapPhase = TRUE,$$
$$MapPhase = TRUE \Rightarrow MapSplit = TRUE.$$

Moreover, to store the intermediate results of separate phases, we introduce a number of variables ($msplit$, map_result, $rsplit$ and $reduce\_result$) that are updated during execution of the corresponding events. The variable updates are also performed according to the formalisation given in Section 2.2. For instance, the variable $msplit$ is introduced to store the result of the *MSplit* phase. After the execution of the MapSplit event, $msplit$ gets the value equal to *MSplit(idata)*.

Machine MapReduce1_m1 **refines** MapReduce1_m0
Variables $idata,\ output,\ msplit,\ MapSplit,\ ...$
Invariants $OutputMapReduce = TRUE \Rightarrow MapSplit = TRUE \wedge$
$MapSplit = TRUE \Rightarrow msplit = MSplit(idata)\ \wedge ...$
MapSplit $\widehat{=}$
when $MapSplit = FALSE$
then $MapSplit := TRUE$
$msplit := MSplit(idata)$
end
OutputMapReduce **refines** OutputMapReduce $\widehat{=}$
any $t2,\ t3,\ t4$
where $MapSplit = TRUE \wedge OutputMapReduce = FALSE \wedge$
$t2 = Map(t1) \wedge t3 = RSplit(t2) \wedge t4 = Reduce(t3)$
with $t1 = msplit$
then $output := Combine(t4)$
$OutputMapReduce := TRUE$
end

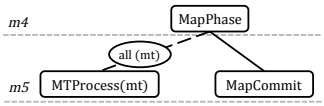

Fig. 4. Blocking model: ERS diagram (for MapPhase)

Breaking atomicity of the Map phase. In the second refinement step, we introduce the event MapPhase that abstractly models the *Map* phase. Essentially, the *Map* phase involves parallel execution of all the map tasks. To introduce such a behaviour, we use the constructor "*all* constructor", which is applied to the MTProcess event that models the execution of a particular map task (see Fig.4). The expression "*all(mt)*" means that the MTProcess event can be enabled for multiple values of $mt \in \mathsf{dom}(msplit)$. On the other hand, the MapCommit event can only occur when all the map computations of map tasks have been finished. In Event-B, we model this by adding a variable $MTProcess$, which is a set containing all possible map tasks that should be processed. The order between the events is ensured by the invariants on the control variables, e.g.,

$$\texttt{inv4:}\ \ MapCommit = TRUE \Rightarrow MTProcess = \mathsf{dom}(msplit),$$

where $\mathsf{dom}(msplit)$ defines the set of all current map tasks. The invariant states that if the MapCommit event has been executed, then all the map tasks have been completed before it. While specifying the MTProcess event, we rely on the definition of the `map` function, given in Section 2.2.

Machine MapReduce1_m5 **refines** MapReduce1_m4
MTProcess $\widehat{=}$
  any mt
  where $MapSplit = TRUE \wedge mt \in dom(msplit) \wedge mt \notin MTProcess$
  then $MTProcess := MTProcess \cup \{mt\}$
    $MTProcess\_result(mt) := map(msplit(mt))$
  end
MapCommit **refines** MapPhase $\widehat{=}$
  when $MapSplit = TRUE \wedge MapCommit = FALSE \ \wedge MTProcess{=}dom(msplit)$
  then $MapCommit = TRUE$
    $map\_result := (\bigcup mt \cdot mt \in dom(msplit) | \{mt\} \times MTProcess\_result(mt))$
  end

Further refinements of the Map phase. During the MapReduce execution, all the map and reduce tasks are parallelised and distributed to multiple processing nodes – the actual software components that carry out the computations. We name these components as *map* and *reduce workers*. Moreover, there is a special component – *master* – that controls all the computations and assigns the map and reduce tasks to the workers. The master periodically pings every worker. In case of a worker failure, the master re-assigns tasks from the failed worker to a healthy one. This procedure can be repeated until the master gets the result for a particular map or reduce task from some worker. To introduce such functionality, we carry out several further refinements focusing on the *Map* phase. These refinements elaborate on modelling of map task execution.

Fig.5 illustrates the event MTProcess and its several consecutive levels of atomicity decomposition. First, the abstract event MTProcess is broken into two concrete events, MTok and MTSuccess correspondingly. The MTok event models the execution of the map task mt by a particular map worker mw. The result

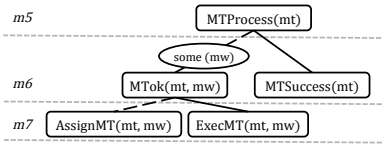

Fig. 5. Blocking model: ERS diagram (for MTProcess)

of this computation should be approved by the master side, which is modelled by execution of the MTSuccess event. The "*some*" constructor indicates that the event MTok may be executed only for some instances of the *mw* parameter before the MTSuccess event becomes enabled. The *MTSuccess* and *MTok* control variables are defined as sets, which allows for multiple executions of the MTSuccess and MTok events. Later on, in the next refinement step, the atomicity of the MTok event is broken into two events AssignMT and ExecMT. The event AssignMT models an assignment of a map task *mt* to a particular map worker *mw*, while ExecMT models the successful execution of the task by this worker.

Similarly to the *Map* phase, we refine the *Reduce* phase by gradually unfolding its computations. The overall refinement structure is presented on Fig.6.

Let us note that the proposed architecture is *blocking* in the sense that the reduce computations can be only started after all the map computations have been finished. The formal derivation of the blocking model and its dynamics is performed under this condition. Next we propose an alternative architectural solution of the MapReduce framework that weakens blocking between the map and reduce stages and, as a result, achieves a higher degree of parallelisation of the MapReduce computations. For this purpose, we will make use of the *dependence relation* between map and reduce tasks introduced in the Section 2.2. We call this model *partially blocking model*.

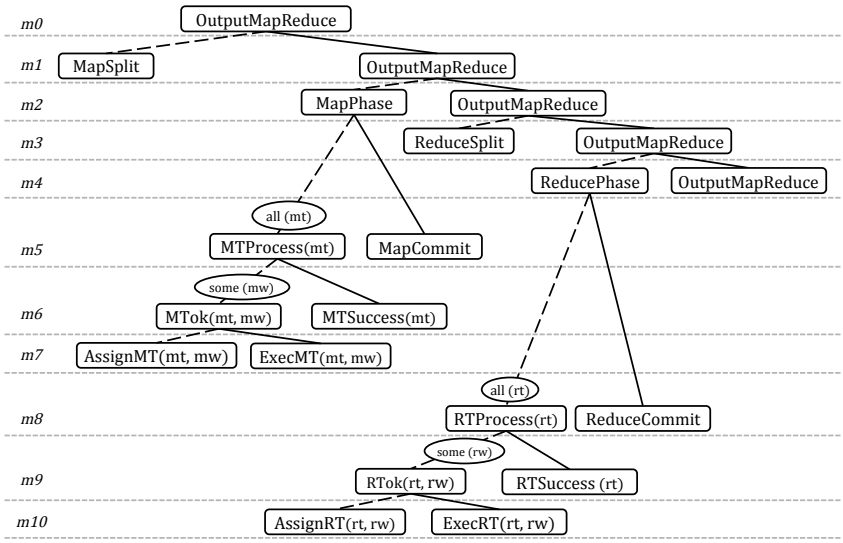

Fig. 6. MapReduce ERS Diagram: blocking model

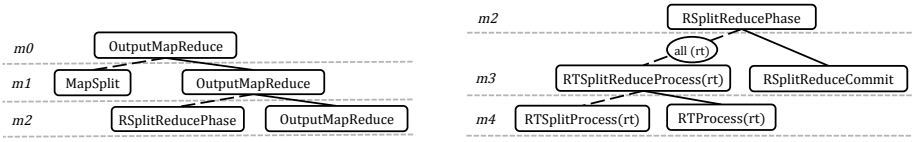

Fig. 7. Partially blocking model: ERS diagrams

4.2 Partially Blocking Model of MapReduce

We start from the same initial specification as for the blocking model, in which the whole MapReduce computation is done in one atomic step, and then refine it in order to introduce the *MSplit* phase. Next, in contrast to the previous derivation, we separate the phase that combines executions of the *RSplit* and *Reduce* phases – RSplitReducePhase. Fig.7 (a) presents the ERS diagram of the refined model.

RSplitReducePhase involves executions of the *RSplit* and *Reduce* phases for all reduce tasks. Essentially, these computations are parallelised. To introduce such behaviour, we use the"*all*" constructor applied to the RTSplitReduceProcess event that, for a particular reduce task rt, performs split and then reduce computations (see. Fig.7 (b)). Next, we separate these split and reduce executions of the particular reduce task rt. Namely, the event RTSplitReduceProcess is split into two concrete events, RTSplitProcess and RTProcess. Here we again rely on the *rsplit* and *reduce* functions formalised in Section 2.2.

Up to now we did not introduce the Map phase explicitly. However, the results of *MapPhase* are simulated internally, by storing the intermediate results in the local variables of the RTSplitProcess event. To explicitly model the Map phase, the event RTSplitProcess is now split into two events MTProcess and RSplit (see Fig.8). The constructor "all" is parameterised by $(mt \in dep[\{rt\}])$". It means that the event MTProcess is executed for all those map tasks, mt, that are in data dependency with the reduce task rt. Therefore, to start the *RSplit* phase, we do not need to wait until all the map tasks are completed. Here we are relying on the definition of data interdependency *dep* between the map and reduce stages, formalised in Section 2.2. Finally, the MTProcess and RTProcess events are refined in the same manner as in the blocking model presented in the Section 4.1.

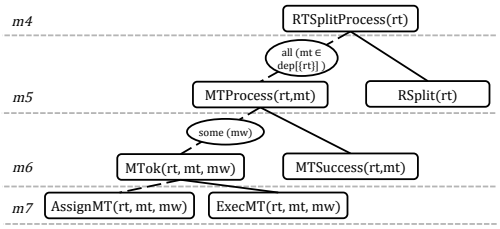

Fig. 8. Partially blocking model: ERS diagram (for RTSplitProcess)

Let us note that the proposed partially blocking model allows us to achieve a higher degree of parallelisation of MapReduce computations. Indeed, for a particular reduce task, when the dependent map tasks have already been executed, the *RSplit* phase for this reduce task can be performed, and then reduce computations can be started. In other words, the computations from three different phases – *Map*, *RSplit*, and *Reduce* – can be performed in parallel, provided the involved data are independent. Therefore, the proposed architectural solution weakens blocking between the stages and, as a result, achieves a higher degree of parallelisation. The overall refinement structure of the partially blocking model is presented on Fig.9.

4.3 Discussion and Future Work

To verify correctness of the presented models, we have discharged around 270 proof obligations for the first formal development, as well as more than 300 for the second one. Approximately 93% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment. With help of the ERS approach, we have decomposed the atomicity of the MapReduce framework and hereby achieved a higher degree of automation in proving. Moreover, the ERS diagrammatic notation has provided us with additional support to represent the model control flow at different abstraction levels and also simplified reasoning about possible refinement strategies. The whole development and proving effort has taken about one person-month.

As a result of the presented refinement chains, we have arrived at two different centralised Event-B models of the distributed MapReduce framework. As a part of the future work, we are planing to derive distributed models by employing the existing decomposition mechanisms of Event-B. This would result in

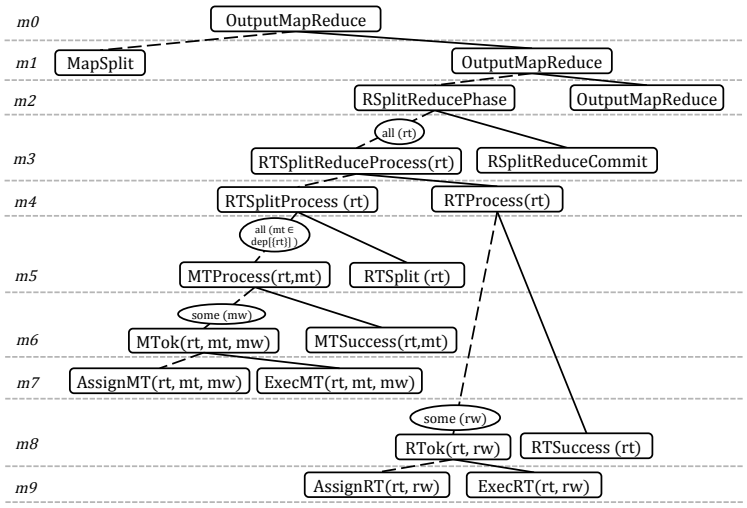

Fig. 9. MapReduce Event Refinement Structure: partially blocking model

creating separate formal specifications of the involved software components of the MapReduce framework (such as master, map worker, reduce worker, etc.).

The static part of the modelled system is formally defined in the corresponding context component. The definitions of static data structures in the context are mostly very abstract, i.e. they state only essential properties to be satisfied. This makes them generic parameters of the whole formal development. In its turn, such formal development becomes generic, representing a family of the systems that can be described by providing suitable concrete values for the generic parameters. The proposed formal model can be used then as a starting point for future development of a specific MapReduce application. The actual concrete values can be supplied by either the end-user (e.g., the `map` and `reduce` functions) or the developer of the MapReduce framework (e.g., the *MSplit* or *RSplit* transformations).

As a continuation of this work, it would be interesting to create formal models for a concrete MapReduce implementation, e.g., the word counting example, by using the Event-B generic instantiation plug-in. Moreover, to analyse the quantitative characteristics of the proposed models, we are planing to use the Uppaal-SMC model checker. This would allow us to, e.g., assign different data processing rates for the map and reduce tasks and then compare the execution time estimations of two considered architectures.

5 Related Work and Conclusions

The problem of formalisation of the MapReduce framework has been studied in [12]. The authors present a formal model of MapReduce using the CSP method. In their work, they focus on formalising the essential components of the MapReduce framework: the master, mapper, reducer, the underlying file system, and their interactions. In contrast, our focus is on modelling the overall flow of control as well as the data interdependencies between the MapReduce computational phases. Moreover, our approach is based on the stepwise refinement technique that allowed us to gradually unfold the complexity of the MapReduce framework.

Formalisation of MapReduce in Haskel is presented in [9]. Similarly to our approach, it focuses on the program skeleton that underlies MapReduce computations and considers the opportunities for parallelism in executing MapReduce computations. However, in addition to that, we also reason about the involved software components – the master, map and reduce workers – that are associated with the respective map and reduce tasks.

The work [7] presents two approaches based on Coq and JML to formally verify the actual running code of the selected Hadoop MapReduce application. In our work we are more interested in formalisation of MapReduce computations and gradual building of different MapReduce models that are correct-by-construction. The performance issues of MapReduce computations have been studied in the paper [4], focusing on one particular implementation of the MapReduce – Hadoop. In contrast, we have tried to formally investigate the data interdependencies between the MapReduce phases and their effect on the degree of parallelisation, independently of a concrete MapReduce implementation.

In this paper we have proposed an approach to formalising the MapReduce framework. Our main technical contribution of this paper is two-fold. On the one hand, based on our definition of interdependencies between the processed data as well as the map and reduce stages, we have derived the conditions under which blocking between the stages can be relaxed. Therefore, we have rigorously derived constraints for implementing MapReduce with a higher degree of parallelisation. On the other hand, we have demonstrated how to use the Event Refinement Structure (ERS) technique to formally derive and verify a model of a complex system with a massively parallel architecture and complex dynamic behaviour.

The stepwise refinement approach to deriving a complex system model has demonstrated good scalability and allowed us to express system properties at different levels of abstraction and with a different degree of granularity. Moreover, combining the refinement technique with tool-assisted mathematical proofs have provided us with a scalable approach to verification of a complex system model.

Acknowledgements. The authors would like to thank the reviewers for their valuable comments. Pereverzeva's work is partly supported by the STV Grant. Butler and Salehis work is partly funded by the FP7 ADVANCE Project (http://www.advance-ict.eu).

References

1. Abrial, J.R.: Modeling in Event-B. Cambridge University Press (2010)
2. Borthakur, D.: The Hadoop Distributed File System: Architecture and Design. In: The Apache Software Foundation (2007)
3. Butler, M.: Decomposition Structures for Event-B. In: Integrated Formal Methods, 7th International Conference, IFM 2009. pp. 20–38. Springer Heidelberg (2009)
4. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M.: MapReduce Online. In: NSDI 2010. pp. 20–20. USENIX Association (2010)
5. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation. pp. 137–150. USENIX Association (2004)
6. Fathabadi, A.S., Butler, M., Rezazadeh, A.: A Systematic Approach to Atomicity Decomposition in Event-B. In: SEFM 2012. LNCS, vol. 7504, pp. 78–93. Springer-Verlag Berlin Heidelberg (2012)
7. Ono, K., Hirai, Y., Tanabe, Y., Noda, N., Hagiya, M.: Using Coq in Specification and Program Extraction of Hadoop MapReduce Applications. In: SEFM 2011. pp. 350–365. Springer Berlin Heidelberg (2011)
8. Pereverzeva, I., Butler, M., Fathabadi, A.S., Laibinis, L., Troubitsyna, E.: Formal Derivation of Distributed MapReduce. Tech. Rep. 1099, TUCS (2014)
9. Ralf Lämmel: Google's MapReduce programming model. vol. 70, pp. 1–30 (2008)
10. Rodin: Event-B Platform, online at http://www.event-b.org/
11. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive - A Warehousing Solution Over a Map-Reduce Framework. In: Proc. VLDB Endowment. vol. 2, pp. 1626–1629 (2009)
12. Yang, F., Su, W., Zhu, H., Li, Q.: Formalizing MapReduce with CSP. In: 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems. pp. 358–367. IEEE computer society (2010)