Decomposition Structures for Event-B

Michael Butler

School of Electronics and Computer Science
University of Southampton, UK
mjb@ecs.soton.ac.uk

Abstract. Event-B provides a flexible approach to modelling and refinement of systems. In this paper we outline two important ways in which Event-B refinement can be augmented with additional structuring to support further the management of complex refinements. Firstly we show how event refinement diagrams can be used to structure refinement steps involving decomposition of atomicity. Secondly we outline a technique for decomposing models into sub-models to allow for independent refinement. We show how these two structuring techniques can be used together.

1 Introduction

An Event-B machine consists of a collection of variables, invariants on those variables and a collection of guarded events that may update the machine variables. An Event-B devlopment consists of a collection of machines linked by refinement.

Event-B [2] provides a more flexible approach to refinement than found in Classical B [1] and in related languages such as Z [11] and VDM [10]. One important feature is the ability to introduce new events in a refinement step. These new events correspond to stuttering steps that are not visible at an abstract level. A very common pattern of Event-B refinement for many types of system, including sequential, concurrent and distributed systems, is to represent a desired outcome as an abstract atomic event and then decompose that into smaller (sub-)atomic steps in refinement. While the Event-B refinement rules are quite comprehensive and allow for decomposition of event atomicity, they are more general than that. By identify a pattern and providing additional structure to represent the pattern, we hope to make the application of the standard refinement rules clearer and more manageable. In this paper we will see how a diagrammatic notation inspired by the structure diagrams of Jackson System Development (JSD) [9] can help to structure refinements involving atomicity decomposition.

Another critical structuring mechanism for refinement is the ability to decompose machines into sub-machines. Typically these sub-models will represent separate archtectural components. We will present a technique for syntactically partitioning an Event-B machine into several sub-machines. This technique has a sound semantic basis that corresponds to the synchronous parallel composition of processes as found in process algebra such as CSP [8]. An important property

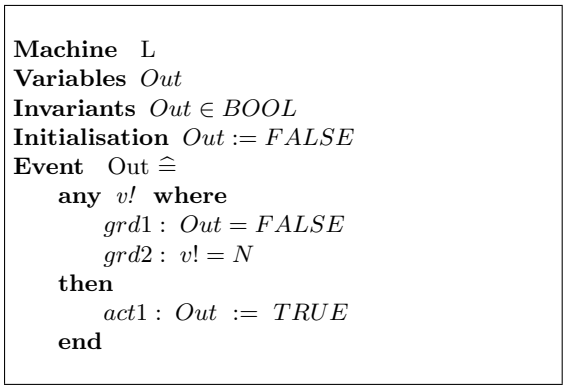


Fig. 1. Abstraction of model of simple outputting machine.

of the decomposition technique is that the resulting sub-models can be refined independently of each other.

2 Decomposing Atomicity

In this section we will look at how coarse-grained atomicity can be refined to more fine-grained atomicity. The approach we take is to treat most of the sub-atomic events of a decomposed abstract event as hidden events which are required to refine *skip*. The new events introduced in a refinement step can be viewed as hidden events not visible to the environment of a system and are thus outside the control of the environment. In Event-B, requiring a new event to refine *skip* corresponds to the process algebraic principle that the effect of an event is not observable. Any number of executions of an internal action may occur in between each execution of a visible action.

Assume we are refining a machine $M1$ by a machine $M2$. In Event-B, each event A of $M2$ either refines some event $R(A)$ of $M1$ or it is a new event refining *skip*. The proof obligations defined for Event-B refinement are based on the following proof rule that makes use of a gluing invariant J:

- Each $M2.A$ (data) refines $M1.R(A)$ under J, if $R(A)$ is defined
- Each $M2.A$ refines *skip* under J, if A is a new event

The machine L of Fig. 1 has a single event Out that simply outputs N and then disables itself. The machine contains a single variable for modelling the control of execution of the Out event: $Out \in BOOL$ is true when the output event has occurred. The Out event can occur provided Out has not occured ($grd1$). The parameter $v!$ represents the output value produced by the Out event. Its value is N ($grd2$).

We wish to refine this machine by a machine modelling a concurrent program that accumulates a value in a variable x before outputting it. The refinement models N parallel subprocesses each of which increments the variable x exactly

once. When all N subprocesses have incremented x, the value of x is output. We view the refined model as breaking the atomicity of the output event by introducing an Inc event that models the behavior of the parallel sub-processes. The decomposition of the atomicity of the simple concurrent program is modelled as an *event refinement diagram* in Fig. 2. This diagrammatic notation is based on JSD structure diagrams by Jackson [9]. The event refinement diagram of Fig. 2 is a tree structure with root $Out(N)$ representing the abstract output event. The diagram shows how the root is decomposed into an initialisation, the parallel composition of multiple parallel instances of $Inc(p)$ and a refined output event $Out(x)$. The oval with the keyword **par** represents a quantifier that replicates the tree below it. In this case it replicates $Inc(p)$ by quantifying over sub-process identifiers p. An important feature of event refinement diagrams, in common with JSD structure diagrams, is that the subtrees are read from left to right and indicate sequential control from left to right. This means that our diagram indicates that the abstract $Out(N)$ event is realised in the refinement by firstly executing the initialisation, then executing the $Inc(p)$ events in parallel (in an interleaved fashion) and then executing $Out(x)$.

Another important feature of event refinement diagrams is the solid and dashed lines linking children to their parent. The $Init$ and $Inc(p)$ events are linked by a dashed line which means it must be proven that they refine *skip*. The abstract and refined Out events are linked by a solid line which indicates a refinement relation. That is, it must be proven that $Out(x)$ refines $Out(N)$.

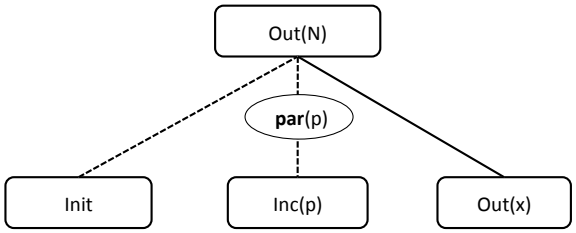


Fig. 2. Event refinement diagram illustrating atomicity decomposition

The refined machine is shown in Fig. 3. It uses a type $PROC$ representing the set of sub-process identifiers with the assumption that $card(PROC) = N$. In addition to the variable x, machine M contains two variables for modelling the control of execution of events. Variable $Inc \subseteq PROC$ represents the set of processes for which the increment event has occurred. Variable $Out \in BOOL$ is true when the output event has occurred. In this case the initialisation of the program is modelled by the standard initialisation clause of the machine M so we do not need a control variable for the initialisation. The Inc event can occur for process p provided Inc has not already occurred for process p. This constraint is modelled by guard $grd1$ of Inc. The action $act1$ of the Inc event

Machine M
Variables $x,\ Inc,\ Out$
Invariants $x \in \mathbb{N},\ \ Inc \subseteq PROC,\ \ Out \in BOOL$
Initialisation $x := 0,\ \ Inc := \{\},\ \ Out := FALSE$
Event Inc $\widehat{=}$
  any p **where**
    $grd1:\ p \in PROC \setminus Inc$
  then
    $act1:\ Inc\ :=\ Inc \cup \{p\}$
    $act2:\ x\ :=\ x+1$
  end
Event Out $\widehat{=}$
  any $v!$ **where**
    $grd1:\ Inc = PROC$
    $grd2:\ Out = FALSE$
    $grd3:\ v! = x$
  then
    $act1:\ Out\ :=\ TRUE$
  end

Fig. 3. Event-B refinement of a simple output machine.

adds the value p to the set Inc which prevents the event occurring for that value of p again. The Out event can occur provided Inc has occurred for all processes ($grd1$) and Out has not occured ($grd2$).

Instead of outputting N the refined Out event outputs the value of x ($grd3$). The proof of the correctness of this refinement relies on the following invariant stating that the value of x is equal to the number of processes that have completed their task:

$$x = card(Inc)$$

Therefore when all N processes have completed, x will have the value N and the correct value will be output. This illustrates how control variables (such as Inc) are useful in gluing invariants, allowing for values of data variables (such as x) to be related to values of control variables.

Consider the case where we have two subprocesses so that $PROC = \{p1, p2\}$ and $N = 2$. The event traces of the model are as follows:

$$\langle\ Inc.p1,\ Inc.p2,\ Out.2\ \rangle \qquad \langle\ Inc.p2,\ Inc.p1,\ Out.2\ \rangle$$

Each event trace represents a record of a possible execution trace of the model. Here we are ignoring the initialisation event since it always occurs exactly once at the beginning of a trace. The parallel execution of the subprocesses is modelled by interleavings of the atomic steps of the processes. Here the two possible interleavings of $Inc.p1$ and $Inc.p2$, represented by the two events traces, model

their concurrent execution. It is instructive to relate the event traces of the machine L with those of machine M. L has just a single event trace that outputs N and nothing else. In the case that $N = 2$, the single event trace of L is

$$\langle\ Out.2\ \rangle$$

If we remove the Inc events from the traces of M we get the trace of L:

$$\begin{array}{lll} \langle\ Inc.p1,\ Inc.p2,\ Out.2\ \rangle\ \setminus\ Inc & = & \langle\ Out.2\ \rangle \\ \langle\ Inc.p2,\ Inc.p1,\ Out.2\ \rangle\ \setminus\ Inc & = & \langle\ Out.2\ \rangle \end{array}$$

Removing events from a trace is the standard way of giving a semantics to hidden or stuttering events and is used, for example, in CSP. By treating the Inc events as a hidden, traces of M look like traces of L. This illustrates a semantics of refinement of Event-B models. Machine M is a refinement of machine L since any trace of M in which the Inc events are hidden is also a trace of L. This is treated more precisely in [5].

3 Decomposing File Write

We will study a further example of atomicity refinement which involves more event interleaving than the simple concurrent program. This is an event for writing a file to a disk. At the abstract level the entire contents of the file is written in one atomic step as in the following machine:

Machine File1
Variables $file,\ dsk$
Invariants $file \subseteq FILE, \quad dsk \in file \rightarrow CONT$

Event Write $\widehat{=}$
any f, c **where**
$grd1:\ f \in file$
$grd2:\ c \in CONT$
then
$act1:\ dsk(f) := c$
end

Here the contents of the disk are represented by the variable dsk which maps file identifiers to their contents. The $Write$ event has two parameters, the identity of the file to be written f and the contents to be written c. Other events such as creating a file and reading a file are not shown.

We assume that file contents are structured as a set of pages of data so that the type $CONT$ is defined as follows:

$$CONT = PAGE \nrightarrow DATA$$

The event refinement diagram of Fig. 4 illustrates the decomposition of the $Write$ event into sub-events to model the writing of individual pages. In the

refinement, the writing of individual pages will be modelled atomically by the $PageWrite$ event and the writing of the entire file is no longer atomic. The writing of a file is initiated by the $StartWrite$ event and ended by the $EndWrite$ event. We will allow multiple file writes to be taking place simultaneously in an interleaved fashion. This is indicated by the top level parallel quantification over f (**par**(f)). We also assume that the pages of an individual file f can be written in parallel hence the inner parallel quantification over p (**par**(p)). Occurrence of event $PageWrite(f, p)$ models writing of page p of file f.

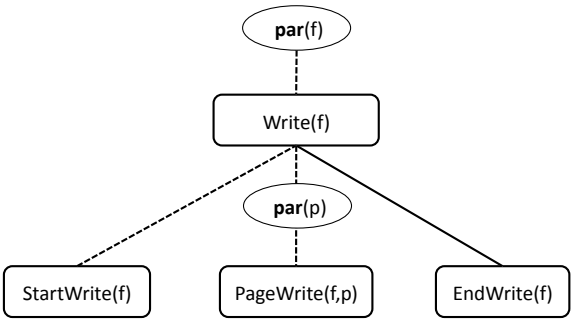


Fig. 4. Decomposition of the atomiticy of file write

In order to model the event sequencing implied by Fig. 4, we introduce variables corresponding to the $StartWrite$ and $PageWrite$ events as follows:

Invariants

$inv1: \ StartWrite \subseteq FILE$
$inv2: \ PageWrite \subseteq FILE \times PAGE$
$inv3: \ dom(PageWrite) \subseteq StartWrite$

The types of these variables are determined by the parallel quantification in Fig. 4. $StartWrite$ is a subset of $FILE$ because it is bound by the quantification over files f ($inv1$). $PageWrite$ is a subset of $FILE \times PAGE$ because it is bound by the quantification over files f and pages p ($inv2$). If a page has been written for a file, then $StartWrite$ will already have occurred for that file ($inv3$).

When the writing of a file is complete, we will allow the file to be written to again. Therefore we do not need any variable to model the occurrence of the $EndWrite$ event for a file, since all the control information for a file will be cleared when the file write is complete in order to allow the file to be written to again later if required. Now, for example, the control behaviour of the $StartWrite$ and $PageWrite$ events is as follows:

Event StartWrite $\widehat{=}$
any f **where**

$grd1:\ f \in file$
$grd2:\ f \notin StartWrite$
then
$act1:\ StartWrite := StartWrite \cup \{f\}$
end
Event PageWrite $\widehat{=}$
any f, p **where**
$grd1:\ f \in StartWrite$
$grd2:\ f \mapsto p \notin PageWrite$
then
$act1:\ PageWrite := PageWrite \cup \{f \mapsto p\}$
end

This control behaviour on its own is not enough. The pages and their contents for a particular file need to be determined before we start the process of writing to a file. We introduce a variable $writebuf$ to act as a buffer for the content to be written to disk. Rather than writing directly to the abstract variable dsk, the $PageWrite$ event will write the contents of an indivdual page to a shadow disk while the writing is in progress. When the writing is complete, the contents of the shadow disk is transferred to the disk at the end of the writing process. These variables are defined as follows:

$inv4:\ writebuf \in StartWrite \rightarrow CONT$
$inv5:\ sdsk \in StartWrite \rightarrow CONT$

Note that both are defined on files that are currently being written, i.e., files in the set $StartWrite$.

Now, as well as initialising the control for the writing process, the $StartWrite$ event sets the contents to be written to disk in the write buffer for that file ($act2$) and sets the shadow disk for that file to be empty ($act3$):

Event StartWrite $\widehat{=}$
any f, c **where**
$grd1:\ f \in file$
$grd2:\ f \notin StartWrite$
$grd3:\ c \in CONT$
then
$act1:\ StartWrite := StartWrite \cup \{f\}$
$act2:\ writebuf(f) := c$
$act3:\ sdsk(f) := \varnothing$
end

The $PageWrite$ event selects a page of a file that has yet to be written ($grd2$) and is in the write buffer ($grd3$). The parameter d represents the data associated with the page being written to the shadow disk ($sdsk$):

Event PageWrite $\widehat{=}$
any f, p, d **where**

$grd1:\ f \in StartWrite$
$grd2:\ f \mapsto p \notin PageWrite$
$grd3:\ p \mapsto d \in writebuf(f)$

then

$act1:\ PageWrite := PageWrite \cup \{f \mapsto p\}$
$act2:\ sdsk(f) := sdsk(f) \Leftarrow \{p \mapsto d\}$

end

The $StartWrite$ and $PageWrite$ events both refine $skip$ while the $EndWrite$ event refines the abstract $Write$ event (see the dashed and solid lines in Fig. 4). The $EndWrite$ event occurs once all pages of a file have been written, a condition that is captured by $grd2$ below. The effect of the event is to copy the shadow disk to the disk ($act1$). The event also clears all the control, buffer and shadow information for the file to enable the write process to commence all over again ($act2$ to $act5$).

Event EndWrite **Refines** Write $\widehat{=}$

any f, c **where**

$grd1:\ f \in StartWrite$
$grd2:\ PageWrite[\{f\}] = dom(writebuf(f))$
$grd3:\ c = sdsk(f)$

then

$act1:\ dsk(f) := sdsk(f)$
$act2:\ StartWrite := StartWrite \setminus \{f\}$
$act3:\ PageWrite := \{f\} \lhd PageWrite$
$act4:\ writebuf := \{f\} \lhd writebuf$
$act5:\ sdsk := \{f\} \lhd sdsk$

end

It may seem like we have not really achieved much decomposition of atomicity since the shadow disk is copied to the actual disk in one atomic step ($act1$ of $EndWrite$). However our intention is that the disk and the shadow together are both realised on the real hard disk and that the effect of $act1$ would be achieved by an update to the page table for the disk (in later refinements). We assume that updating the page table can reasonably be treated as atomic. Having the $PageWrite$ event write the individual pages to a shadow disk also allows us to model fault tolerance quite easily. We add an $AbortWrite$ event that clears all the control and shadow information for a file write but does not update the disk:

Event AbortWrite $\widehat{=}$

any f **where**

$grd1:\ f \in StartWrite$

then

$act1:\ StartWrite := StartWrite \setminus \{f\}$
$act2:\ writebuf := \{f\} \lhd writebuf$
$act3:\ sdsk := \{f\} \lhd sdsk$
$act4:\ PageWrite := \{f\} \lhd PageWrite$

end

This event refines *skip* since it does not modify the *dsk* variable that appears in the abstract model. Thus the effect of an abort, which can happen after any number of pages are written, is to leave the disk in the state it was in before the file write process started (for the file f).

It is instructive to compare an event trace of the abstract file model with a corresponding trace of the refinement file model. The following trace represents a behaviour in which the contents $c2$ is written to file $f2$ and then the contents $c1$ is written to file $f1$:

$$\langle\ Write.f2.c2,\ \ Write.f1.c1\ \rangle$$

Each of these high-level events is realised by several new events ($StartWrite$, $PageWrite$ etc). The sub-events of one high-level write may interleave with those of the other high-level event. For example, the following event trace of the refined model illustrates this (the events that directly refine an abstract event are highlighted in bold):

$$\begin{aligned}&\langle\ StartWrite.f1.c1,\ \ PageWrite.f1.p1.c1(p1),\\ &StartWrite.f2.c2,\ \ PageWrite.f1.p2.c1(p2),\\ &PageWrite.f2.p1.c2(p1),\ \ PageWrite.f2.p2.c2(p2),\\ &\textbf{EndWrite.f2.c2},\ \ PageWrite.f1.p3.c1(p3),\ \ \textbf{EndWrite.f1.c1}\ \rangle\end{aligned}$$

This illustrates a scenario in which writing to file $f1$ is started before writing to $f2$ is started but writing of file $f2$ finishes before writing of file $f1$.

To recap, we have decomposed the atomicity of the abstract $Write$ event by introducing the new events $StartWrite$, $PageWrite$ and $AbortWrite$ and by refining the $Write$ event with the $EndWrite$ event. Formally, the new events have no connection to the abstract $Write$ event, only the $EndWrite$ has a formal connection. However, the event refinement diagram of Fig. 4 describes the intended purpose of the new events which is to represent the intermediate steps of the file write process that lead to a state where the $EndWrite$ is enabled. The diagram also plays another role in that it defines the control behaviour of all the events constituting the write process and this was encoded in Event-B in a systematic way, i.e., introducing the $StartWrite$ and $PageWrite$ control variables. The additional modelling elements provided, $writebuf$ and $sdsk$, were required in order to model abstractly the effect of the various events and their introduction was based on modelling judgement.

4 Decomposing machines

In this section, we describe a parallel composition operator for machines. The parallel composition of machines M and N is written $M \parallel N$. Machines M and N must not have any common state variables. Instead they interact by synchronising over shared events (i.e., events with common names). They may

also pass values on synchronisation. We look first at basic parallel composition and later look at parallel composition with shared parameters. We show how the composition operator may be applied in reverse in order to decompose system models into subsystem models.

In general, an event has the form

$$\textbf{any } x \textbf{ where } G \textbf{ then } S \textbf{ end}$$

where x is a list of event parameters, G is a list of guards (implicitly conjoined) and S is a list of actions on the machine variables (implicitly simultaneous). We write $G \wedge H$ to join two lists of guards and $S \parallel T$ to join two lists of actions.

To achieve the synchronisation effect between machines, shared events from M and N are 'fused' using a parallel operator for events. Assume that m (resp. n) represents the state variables of machine M (resp. N). Variables m and n are disjoint. The parallel operator for events is defined as follows:

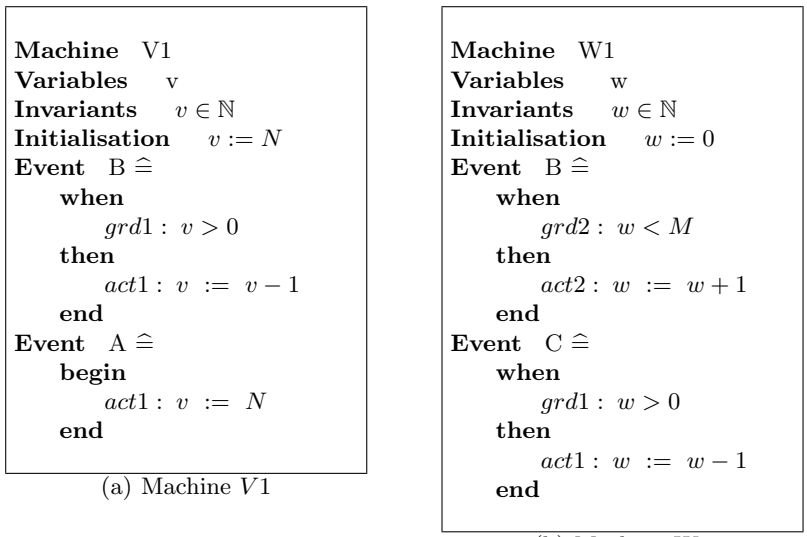
$$\begin{array}{rcl} ev1 & = & \textbf{any } y \textbf{ where } G(y,m) \textbf{ then } S(y,m) \textbf{ end} \\ ev2 & = & \textbf{any } z \textbf{ where } H(z,n) \textbf{ then } T(z,n) \textbf{ end} \end{array}$$

$$\begin{array}{rcl} ev1 \parallel ev2 & \widehat{=} & \textbf{any } \ y,z \ \textbf{ where} \\ & & \quad G(y,m) \wedge H(z,n) \\ & & \textbf{then} \\ & & \quad S(y,m) \parallel T(z,n) \\ & & \textbf{end} \end{array}$$

The parallel operator models simultaneous execution of the actions of the events and the composite event is enabled exactly when both component events are enabled. This models synchronisation: the composite system engages in a joint event when both systems are willing to engage in that event. The parallel composition of machines M and N is a machine constructed by fusing shared events of M and N and leaving independent events independent. The state variables of the composite system $M \parallel N$ are simply the union of the variables of M and N.

As an illustration of this, consider machines *V 1* and *W 1* of Fig. 5. The machines work on independent variables v and w respectively. Both machines have an event labelled B and to compose these machines we fuse their respective B events. The composition of both machines is shown in Fig. 6. The A event and C event of *VW 1* come directly from *V 1* and *W 1* respectively as they are not joint events rather they are independent events. The B event is a joint event and is defined as the fusion of the B-events of *V 1* and *W 2*. The initialisations of *V 1* and *W 1* are also combined to form the initialisation of *VW 1*. The joint B event simultaneously decreases v while increasing w, provided $v > 0$ and $w < N$.

We have presented *VW 1* as having been formed from the composition of *V 1* and *W 1*. We can view the relationship between these machines in another way. Let us suppose we had started with *VW 1* and decided that we wish to decompose it into subsystems. The diagram in Fig. 7(a) illustrates the dependencies between events and variables in the machine *VW 1*. For example, the line from the box



(a) Machine $V1$

(b) Machine $W1$

Fig. 5. Machines to be composed in parallel

indicating event A to the oval indicating variable v represents the fact that event A depends on v, i.e., it may read from and assign to v. The diagram shows that B is the only event that depends on both v and w suggesting that B needs to be a shared event if we are to partition v and w into separate subsystems. This decomposition is illustrated in Fig. 7(b) where variables v and w of $VW\,1$ are partitioned into subsystems $V\,1$ and $W\,1$ respectively, A is an event of subsystem $V\,1$, C is an event of subsystem $W\,1$ and B is an event shared by both subsystems.

The B event of system $VW\,1$ is partitioned into two parts, one of which will belong in $W\,1$ and the other in $W\,1$. The B event has an important characteristic that allows it to be partitioned in this way. The guards and actions depend either on v or on w but not both. So, guard $grd1$ and action $act1$ both depend on v only, while guard $grd2$ and action $act2$ both depend on w. This localisation of variable dependency allows us to easily partition the guards and actions of the B event of $VW\,1$ into the separate B events of $V\,1$ and $W\,1$ respectively.

We extend the fusion operator to deal with shared event parameters. Events to be fused must depend on disjoint machine variables but they may have common parameters and these common parameters are treated as joint parameters in the fused event. In the following, x represents parameters that are joint across events and y and z are local to their respective events:

$$\begin{aligned} ev1 &= \textbf{any } x, y \textbf{ where } G(x,y,m) \textbf{ then } S(x,y,m) \textbf{ end} \\ ev2 &= \textbf{any } x, z \textbf{ where } H(x,z,n) \textbf{ then } T(x,z,n) \textbf{ end} \end{aligned}$$

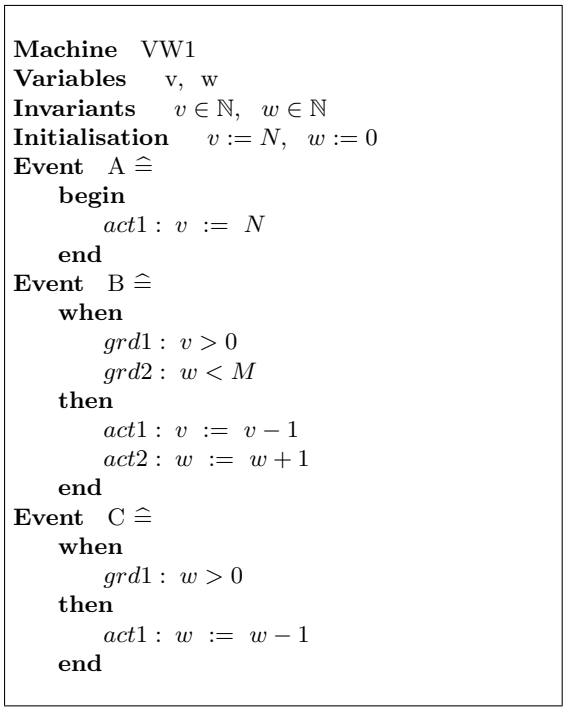
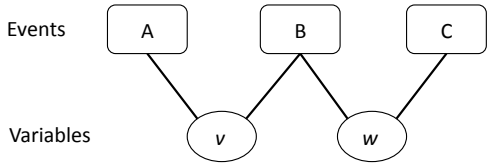


Fig. 6. Composition of $V1$ and $V2$.

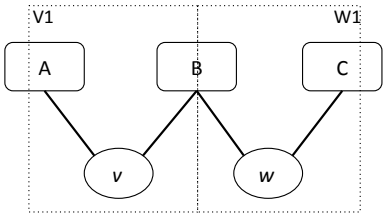
$$
\begin{array}{lll}
ev1 \parallel ev2 & \widehat{=} & \textbf{any} \quad x, y, z \quad \textbf{where} \\
 & & \quad G(x, y, m) \wedge H(x, z, n) \\
 & & \textbf{then} \\
 & & \quad S(x, y, m) \parallel T(x, z, n) \\
 & & \textbf{end}
\end{array}
$$

We illustrate the use of shared parameters by extending the *VW1* machine slightly. Assume that instead of increasing v and decreasing w by 1 in the B event, we modify both v and w by a value i. To do this we give the B event a parameter i which is used to modify the variables as follows:

$$
\begin{array}{l}
\textbf{Event} \quad \text{B} \mathrel{\widehat{=}} \\
\quad \textbf{any} \ \text{i} \ \textbf{where} \\
\quad\quad grd1 : \ 0 \leq i \leq v \\
\quad\quad grd2 : \ w < N \\
\quad \textbf{then} \\
\quad\quad act1 : \ v := v - i \\
\quad\quad act2 : \ w := w + i \\
\quad \textbf{end}
\end{array}
$$



(a) Variable access by events in VW



(b) Split events and variables

Fig. 7. Illustration of decomposition a machine

Now we partition the guards and events of B into those that depend on v and those that depend on w giving the following events:

Event B $\widehat{=}$
any i **where**
$grd1: \ 0 \leq i \leq v$
then
$act1: \ v := v - i$
end

Event B $\widehat{=}$
any i **where**
$grd1: \ i \in \mathbb{Z}$
$grd2: \ w < N$
then
$act1: \ w := w + i$
end

The shared parameter i means that both of these events will agree on the amount by which v and w are respectively decreased and increased. In the left hand sub-event, the guard $grd1$ constraints the value of the parameter based in the state variable v. In the right-hand sub-event, the value of i is not constrained other than a typing guard ($i \in \mathbb{Z}$). This means that the left-hand sub-event can be viewed as outputting the value i while the right-hand sub-event accepts the value i as an input.

When we decompose a system into parallel subsystems, the subsystems may be refined and further decomposed independently. This is a major methodological benefit, helping to modularise the design and proof effort. The semantic justification for this is outlined in [5].

5 Incremental development of a distributed file transfer

In this section we outline an incremental development of a simple system for copying a file from one location to another. The development makes use of event decomposition and machine decomposition. We start with an abstract model in which the file copy occurs in one atomic step. We then refine this by a model in which the contents of the file is copied one page at a time. The refined model is then decomposed into subsystems. Instead of decomposing into two subsystems that synchronise with each other, we decompose into three subsystems as illustrated in Fig. 8. In this decomposition the two agents do not synchronise directly with each other. Instead they interact indirectly through a middleware subsystem. Each agent synchronises directly and separately with the middleware and this will be used to model asynchronous communication between the agents. This form of asynchronous communication via middleware can be used to model many distributed systems that are based on message passing. In order to be able to decompose in this way, we will need to apply refinement steps that enable the agents to be decomposed into asynchronous subsystems.

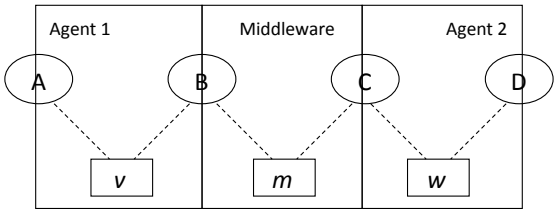


Fig. 8. Decomposition with asynchronous middleware

5.1 Abstract model

The model makes use of the types $PAGE$ and $DATA$ respectively. A file is modelled as a partial function from pages to data. Machine $F1$ defines the abstract behaviour of the file transfer system. It contains two variables $fileA$, representing the contents of the file at the sending side, and $fileB$ representing the value of the file at the receiving side:

Machine F1
Variables *fileA* , *fileB*
Invariants
$inv1: \ fileA \in PAGE \pfun DATA$
$inv2: \ fileB \in PAGE \pfun DATA$

The abstract machine has one event that simply copies the contents of $fileA$ to $fileB$ in one atomic step:

Event CopyFile $\widehat{=}$
begin
$act1: \ fileB := fileA$
end

5.2 Breaking atomicity

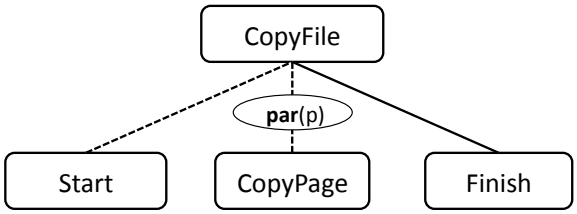


Fig. 9. Refining atomicity of the *CopyFile* event

The atomicity of the *CopyFile* event is decomposed in the same way in which the atomicity of the *Write* event was decomposed in Section 2. This is illustrated in Fig. 9. We introduce control variables based on this diagram as well as a buffer *buf* in which pages are written one at a time by the *CopyPage* event. Further details of this refinement may be found in [5].

5.3 Split events to A side and B side

Before decomposing the file transfer system into three subsystems, we must first split some events into an A-part, representing behaviour on the sending side, and a B-part, representing behaviour on the receiving side. This is illustrated by the diagram in Fig. 10 which shows that the *Start* event is decomposed into *StartA* and *StartB*. The *StartA* event represents the sending side deciding to commence the transfer while the subsequent *StartB* event represents the receiving side recognising that the transfer has commenced. The *StartA* event will set a flag *StartA* to *TRUE* while the *StartB* event will set a flag *StartB* to *TRUE* provided *StartA* is true. The *CopyPage* event is decomposed into separate A and B parts in a similar way. We assume that the sending side will send the size of the file at the start so that the receiving side can know when all the pages have been received. This means that the sending side does not need to send a finish message so we need a *Finish* event on the receiving side only.

The event refinement diagram in Fig. 10 provides a hierarchical overview of the major refinement steps involved in this development so far. The top level corresponds to the abstract atomic event, the intermediate level corresponds to the first refinement where the atomicity of the copy is decomposed and the third level of the hierarchy shows how events are split into two parts for sender and receiver.

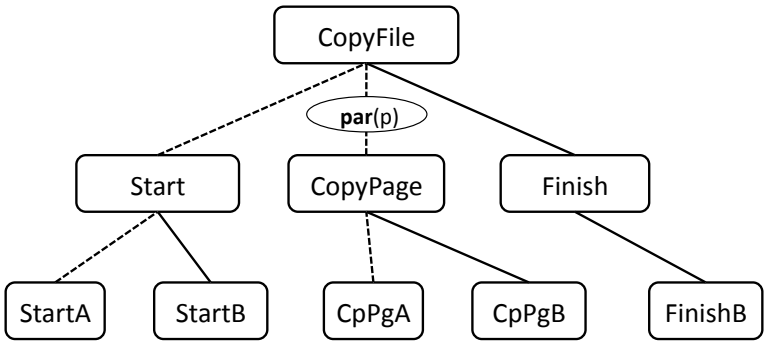


Fig. 10. Splitting events into sender and receiver parts

5.4 Introduce message variables

Now consider again the $StartB$ event just outlined. Our intention is that this is an event of the receiving side so we wish to make it an event of the receiver subsystem. This means it should not refer to variables of the sending side directly since we are aiming at an asynchronous decomposition. However the $StartB$ event does refer to variables of the sending side: for example it refers to the $StartA$ control variable.

To break this dependency on variables of the sending side in events of the receiving side, we introduce variables that duplicate the variables of the sending side, e.g., $StartM$ and $CopyPageM$. These duplicate variables will be separated into a middleware machine (Fig. 8) and become abstract representations of messages in transit in the middleware.

5.5 Separate machines

The previous model is decomposed into three separate machines representing three subsystems as illustrated in Fig. 8. The three machines are:

- machine $mA1$ representing a model of the sending agent
- machine $mB1$ representing a model of the receiving agent
- machine $mM1$ representing a model of the middleware through which the sender and receiver interact.

The variables of the previous model are partitioned amongst the three machines. The sender interacts with the middleware through synchronisation over actions ($StartA$ and $CopyPageA$). Similarly, the receiver interacts with the middleware through synchronisation over actions ($StartB$ and $CopyPageB$). There is no direct interaction between the sender and receiver - all communication is via the middleware machine.

Fig. 11 provides an architectural overview of the decomposition illustrating how the variables and events are distributed amongst the subsystems. The variables allocated to each subsystem are listed in italic in the relevant box for that

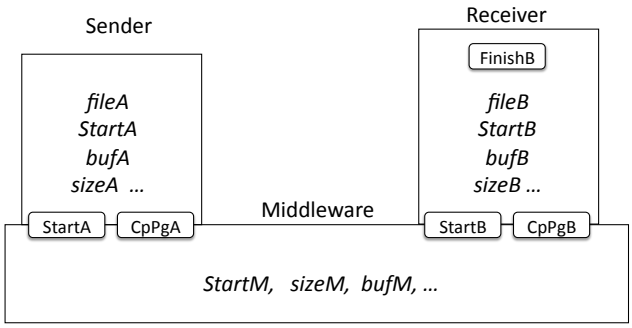


Fig. 11. Architectural illustration of decomposition

subsystem, e.g., the sender subsystem contains the variables $fileA$, $StartA$ etc. The smaller labelled boxes indicate the synchronised shared events. For example, the $StartA$ event is shared between the sender and the middleware representing a synchronised interaction between these subsystems.

See [5] for further details of how the event specifications are decomposed into the separate syntactic components in order to decompose the model. [5] also outlines how the abstract model of the middleware may be refined further so that more explicit datatypes representing messages are introduced reflecting the usual interface to a communications middleware.

6 More about Event Refinement Diagrams

In the event refinement diagrams shown so far, the refining event is always the final step of an event decomposition. For example, in Fig. 2, the refined $Out(N)$ event is the final step in the decomposition of the abstract $Out(N)$ event. It is not a requirement that the refining event always be the final event of a decomposition. Fig. 12 shows an event refinement diagram for an update of a replicated database in which the refining event is followed by further new events. This diagram is based on the structure of a refinement presented in [12] (although event refinement diagrams are not used in [12]). The outline of this development is as follows. The abstract machine models a single database. The refined machine models a set of sites each of which holds its own copy of the database. In the abstract machine, an update of the database is a simple atomic event. The refinement uses a two-phase commit protocol (with *precommit* then *commit* phases) to ensure a consistent distributed update transaction. The phasing is represented in Fig. 12. Once an update transaction t is started, each site s independently precommits to the transaction (which locks all the database objects involved in the transaction). Once all sites have precommitted, the transaction is globally committed by a coordinator. The *GlobalCommit* refines the abstract *Update* since a global decision has been made to update all copies of the database.

After the global commit, each site s locally commits its copy of the database independently (and releases any objects locked by its precommit).

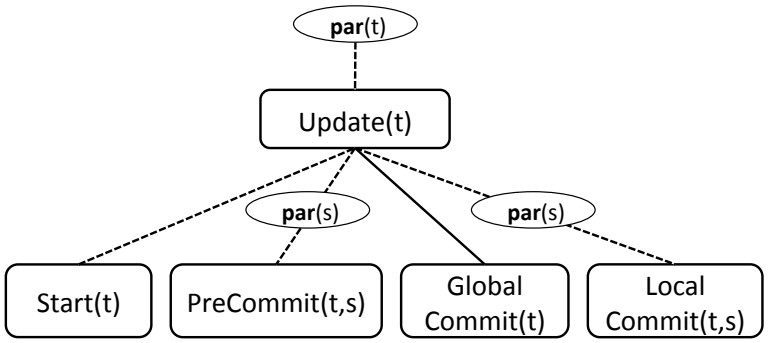


Fig. 12. Event refinement diagram for replicated database update

In this paper we have avoided providing a systematic definition of event refinement diagrams and their translation to Event-B. The reason for this is simply that the concepts are not fully mature at the time of writing. It may be that a complete set of translation rules is not appropriate and that instead a common set of patterns can be identified and translations provided for those. The diagrams seem to be a promising way of representing reusable patterns of event decomposition. They are abstract and visual and humans are good are recognising visual patterns. This is one reason why we have avoided cluttering the diagrams too much with, for example, event guard. Too much clutter may make patterns appear less general.

Our initial exploration of JSD structure diagrams as a means of representing the structure of atomicity decomposition was influenced by the work of Ball [4] on the use of KAOS [6] goal diagrams for a similar purpose. Our event refinement diagrams are different in construction to the refinement diagrams developed by Back [3]. Back's diagrams expose the containment and refinement relationships between general components and subcomponents. In Back's diagrams, enclosing components may be replicated in order to simultaneously illustrate refinements between subcomponents and between enclosing components. In our diagrams the higher level events can be viewed as enclosing components and these only appear once at the top level. Back's diagrams are neutral with respect to the operator used to compose components. In our diagrams the operators (sequential and parallel) are built in.

7 Concluding

We have outlined techniques for atomicity decomposition and machine decomposition. The atomicity decomposition technique uses the standard Event-B re-

finement rule together with event refinement diagrams to provide an explicit representation of the the sequencing of sub-events and the refinement relationships involved. These diagrams provide a systematic means of introducing control structure in an incremental manner through diagram hierarchy. They provide a useful hierarchical overview of multiple refinement steps. They provide a convenient mechanism for exploring several levels of event decomposition in advance of construction of the appropriate Event-B refinements. They also appear to provide a convenient way of representing reusable patterns of event refinements. The machine decomposition technique is based on synchronisation between machines over shared events with asynchronous decomposition as a special case involving an explicit representation of an asynchronous communications medium. The decomposition approach supports independent refinement and decomposition of sub-machines. Together, the event decomposition and machine decomposition techniques augment Event-B by making the application of refinement more systematic and scalable then the standard refinement rules on their own.

Acknowledgements: The work described here is part of the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

References

1. J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.
2. J.-R. Abrial. *Modelling in Event-B: System and Software Engineering.* To be published by Cambridge University Press, 2008.
3. Ralph-Johan Back. Refinement diagrams. In J. M. Morris and R. C. Shaw, editors, *Proceedings of the 4th Refinement Workshop*, pages 125–137, Cambridge, UK, Jan 1991. Springer-Verlag.
4. Elisabeth Ball. An Incremental Process for the Development of Multi-agent Systems in Event-B, PhD thesis, University of Southampton, `http://eprints.ecs.soton.ac.uk/16575/`, August 2008.
5. Michael Butler. Incremental design of distributed systems with Event-B, November 2008. Marktoberdorf Summer School 2008 Lecture Notes.
6. Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, 1993.
7. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice–Hall, 1985.
8. M.A. Jackson. *System Development.* Prentice–Hall, 1983.
9. C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall International, second edition, 1990.
10. J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.
11. D.S. Yadav and M.J. Butler. Formal development of fault tolerant transactions for a replicated database using ordered broadcasts. In *Methods, Models and Tools for Fault Tolerance (MeMoT 2007)*, pages 33–42, May 2007.