Incremental Design of Distributed Systems with Event-B

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

Abstract.
It is shown how Event-B can be use to model and reason about distributed systems from a high-level global view down to a detailed distributed architectural view. It is shown how refinement and decomposition can be used to introduce distribution of state and control and to introduce message passing between components. Distribution is treated as a special case of concurrency. Techniques are presented for decomposing abstract atomic events into smaller atomic steps in refinement and for decomposing models into sub-models.

Keywords. set theory, refinement, invariants, proof obligations, distributed systems, message-passing

1. Introduction

These lectures notes make use of Event-B [1] for modelling and refinement and make use of the Rodin toolset for Event-B [2]. The notes assume some knowledge of the Event-B language, refinement in Event-B, invariants and proof obligations.

It will be shown how Event-B can be use to model and reason about distributed systems from a high-level global view down to a detailed distributed architectural view and will be shown how refinement and decomposition can be used to introduce distribution of state and control and to introduce message passing between components. We treat distribution as a special case of concurrency where the only shared variables are buffers used for message-passing. This is very convenient as it allows us to reason about key properties of systems using simpler global abstractions of state and then refine these to distributed systems. We will look at how one can model concurrency in Event-B by modelling the atomic steps that take place in a concurrent system. We will also look at atomicity refinement whereby an atomic step at an abstract level is decomposed into several smaller atomic steps. This involves refining coarse-grained atomicity with more fine-grained atomicity.

We will look at how a model may be decomposed into sub-models. Typically these sub-models will represent separate archtectural components. We will present a technique for syntactically partitioning an Event-B model into several sub-models. This technique has a sound semantic basis that corresponds to the synchronous parallel composition of processes as found in process algebra such as CSP [13]. An important property of the

decomposition technique is that the resulting sub-models can be refined independently of each other. Our decomposition tehnique will be used to partition the behaviour of agents in a distributed network into separate models, including separate models of message-passing mechanisms.

Performing refinement in incremental steps means that the abstraction gap between refinement levels is not too great for feasible reasoning (formal and informal). This means that the proof effort can be factored out into many relatively simple steps. Simple proof steps allow for a high degree of automation in proof. More automated proof makes it easier to change models

A completed refinement chain (or tree) is usually presented in a top-down manner. However, construction of a refinement chain is rarely top-down. There are several reasons for this. One is that requirements change. Another is that when proving refinement between two models, say $M1$ and $M2$, it may be more convenient to find an intermediate model $M3$ lying between $M1$ and $M2$ in order to simplify the proof effort. A further reason is that our abstract model my turn out to be inaccurate. That is when proving that $M1$ is refined by $M2$, we encounter proof failure and realise that the problem is with $M1$ rather than $M2$. Our understanding of the system changes (improves) as we elaborate the design.

A key ingredient in performing refinement proofs is the gluing invariant linking states of abstraction levels. A key role of the proof obligations generated by the Rodin tool is to verify the maintainence of gluing invariants. But the tool can also be used to help in the *discovery* of appropriate gluing invariants.

A link between modelling in Event-B and modelling with process algebra such as CSP will be made. Typically in process algebra the behaviour of a process is defined in terms of the events in which it can engage. A similar view can be taken of Event-B models and has a bearing on the way in which interaction, composition and refinement are treated. The relationship to modelling and proof in Event-B will be outlined.

In Event-B, a system is specified as an abstract machine consisting of some state variables and some events (guarded actions) acting on that state. This is essentially the same structure as an action system [4] which describes the behaviour of a parallel reactive system in terms of the guarded actions that can take place during its execution. Techniques for refining the atomicity of operations and for composing systems in parallel have been developed for action systems and such techniques are important for the development of parallel/distributed systems. Different views as to what constitutes the observable behaviour of a system may be taken. In the state-based view, the evolution of the state during execution is observable but not the identity of the operations that cause the state transitions. In the event-based view, the execution of an operation is regarded as an event, but only the the identity of the event is observable and the state is regarded as being internal and not observable. The event-based view corresponds to the way in which system behaviour is modelled in process algebras such as CSP [13]. An exact correspondence between action systems and CSP was made by Morgan [15]. Using this correspondence, techniques for event-based refinement and parallel composition of action systems have been developed in [7,8]. In these notes, we shall use the event-based view of action systems, applying the techniques of [7,8] to Event-B machines. For a description of the state-based view of action systems see [5].

Note that in this paper we only deal with preservation of safety properties in refinement. We avoid treating preservation of liveness in the form of convergence of hidden

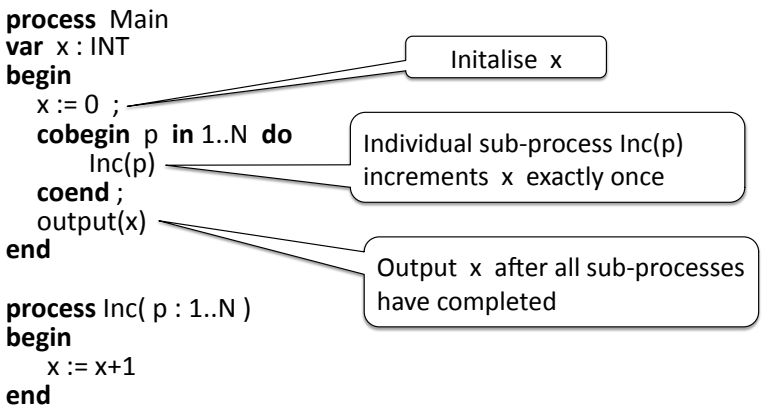


Figure 1. Simple concurrent program with atomic steps identified

events (absence of divergence) and preservation of event enabledness (absence of deadlock). Further discussion of the proof obligations needed for this may be found in [12].

2. An example of treating concurrency in Event-B

In this section we outline how Event-B can be used to model a concurrent system. An Event-B machine consists of some state variables, a set of events, each with its own unique name, and an initialisation action. A machine proceeds by firstly executing the initialisation. Then, repeatedly, an enabled event is selected and executed. A system deadlocks if no event is enabled.

Our approach to modelling concurrency in Event-B is straightforward. We identify the atomic steps that can take place in the system being modelled and make these events. We introduce appropriate state variables to control the order of execution of our chosen events and appropriate state to represent the purpose of the system.

Figure 1 presents a very simple concurrent program written in a fairly conventional structured programming notation (note this is not Event-B). The process consists of a main process $Main$ and N subprocesses $Inc(p)$ where p ranges between 1 and N. The program has a single shared variable x that is initialised to 0. Each subprocess increments x exactly once. The execution of an individual assignment $x := x + 1$ by a subprocess is assumed to be atomic. Once each subprocess completes its simple task, the main process outputs the value of x. Despite its simplicity, we can ask some important questions about this program:

- What does this program achieve?
- Why does it work?
- How would we verify that it works?

We will address these questions by building an Event-B model of the concurrent program. In order to do this we identify the atomic steps that can take place in the program. The callout comments in Figure 1 identify these steps. The initialisation of x is an atomic step. Each subprocess executes a single atomic step when it increments x. The

Context C
Sets $PROC$
Constants N
Axioms

$axm1 : finite(PROC)$
$axm2 : N = card(PROC)$

Figure 2. Context for program model.

final atomic step is to output x. So there are $N + 2$ atomic steps . Rather than having a separate event for each subprocess $Inc(p)$, we will use a single event, parameterised by p, to model the atomic steps of the subprocesses. So our Event-B model will have three events: the initialisation, Inc and Out. This may be seen in the machine M in Figure 3. When modelling multiple instances of some entity (such as a subprocess) in Event-B, it is convenient to use a given type to identify instances. Figure 2 shows the context for our model. It defines a set $PROC$, to represent subprocess instances and a constant N. The set $PROC$ is assumed to be finite with cardinality 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 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 ($grd2$). The parameter $v!$ represents the output value produced by the Out event.

Interaction and control

We see in machine M two different uses of event parameters. In the Inc event, the parameter p is used to identify the process whose atomic step the event occurrence is modelling. In the Out event, the parameter $v!$ is being used to represent an output value produced by an atomic step. We treat these differently in how we interpret the model but we do not treat them differently in proof.

Machine M also illustrates a convention we will adopt about variables for controlling event execution. When we want to limit the occurrence of an event we will introduce a variable with the same name as the event, e.g., Inc and Out in M. If we want a single execution of one instance of the event, we specify the control variable for that event to be of type boolean, initially set to $false$, and model the control as in the Out event of machine M. If we require multiple instances of execution of the event we specify the control variable to be a set of some type T, where T is a type used to identify instances, e.g., $Inc \subseteq PROC$ in M. Execution of such an event with instance value i will add the value of i to the set to indicate that the instance has occurred and to prevent it from occurring again. This is illustrated in the Inc event of machine M.

Machine M **Sees** C
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

Figure 3. Event-B machine for the simple concurrent program.

Event traces of the model

In the CSP process algebra, process behaviour can be given a formal semantics in terms of traces of obervable events of a process. We can do the same with our model of the simple concurrent program. 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. The event traces provide a definition of the observable behaviour of the model using an interleaving semantics. We will treat this more precisely in Section 3.

Machine L **Sees** C
Variables Out
Invariants $Out \in BOOL$
Initialisation $Out := FALSE$
Event Out $\widehat{=}$
any *v!*
where
$grd1 : Out = FALSE$
$grd2 : v! = N$
then
$act1 : Out := TRUE$
end

Figure 4. Abstraction of model of simple concurrent program.

Abstract model of the desired behaviour

We have yet to give a precise answer to the question about what the simple concurrent program achieves. The informal answer of course is that it outputs the value N. We can specify this formally using an Event-B model that outputs the value N. This is represented by the machine L of Figure 4 which has a single event Out that simply outputs N and then disables itself. This model abstracts away from the subprocesses that contribute towards achieving the effect and does not include the variable x that is used to accumulate the value contributed by the subprocesses.

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

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

Recall that the event traces of machine M were

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

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

$$\langle\, Inc.p1,\ Inc.p2,\ Out.2 \,\rangle \setminus Inc \;=\; \langle\, Out.2 \,\rangle$$
$$\langle\, Inc.p2,\ Inc.p1,\ Out.2 \,\rangle \setminus Inc \;=\; \langle\, Out.2 \,\rangle$$

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. We will treat this more precisely in Section 3.

Refinement proof

When both machines L and M, with M declared to be a refinement of L, are given to the Rodin tool several proof obligations are generated and discharged. One proof obligation turns out not to be provable:

$$
\begin{array}{ll}
N = card(PROC) & \text{// from context } C \\
Inc = PROC & \text{// guard } grd1 \text{ of } Out \text{ in } M \\
\vdash & \\
x = N &
\end{array}
$$

This proof obligation is required to ensure that the output value x produced by the refined Out event is the same as the output value N produced by the abstract Out event. The proof obligation lists two hypotheses and a single goal $x = N$. The task is to prove the goal under the hypotheses. Unfortunately the goal is not provable under the given hypotheses since they say nothing about x. To overcome this we need to add an invariant to the model that will be available as a hypothesis in the proof and will be sufficient to prove the goal. The proof obligation gives us a clue as to what the invariant should be. A standard heuristic for constructing an invariant from a goal is to replace a constant by a variable [11]. The hypotheses above allow us to replace the constant N by $card(PROC)$ and then $PROC$ by Inc to get the invariant:

$$x = card(Inc)$$

The expression $card(Inc)$ represents the number of subprocesses that have completed their task (which is to increment x) so this invariant specifies that the value of x is equal to the number of processes that have completed their task. Therefore when all N processes have completed, x will have the value N and the correct value will be output. By correct we mean, of course, the value specified in the abstract model L. When the above invariant is added to the model, all proof obligations are discharged and we are done with proof.

We now have precise answers to the quesions posed at the beginning of this section:

- What does this program achieve? It outputs the value N as specified in the model L of Figure 4.
- Why does it work? Because it is always the case that $x = card(Inc)$ (the invariant).
- How would we verify that it works? By discharging all proof obligations associated with showing that model M refines model L.

An important observation is that the verification has helped us uncover why the program works by forcing us to discover invariants that are sufficient to discharge the proof obligations.

3. Behaviour and Refinement

In this section we will be more precise about trace behaviours of machines and its connection with refinement. A trace is a sequence of *event labels*, where an event label is of

the form $ev.i$ consisting of event name ev and parameter value i. For example, $Inc.p1$ is an event label with event name Inc and event parameter $p1$. We can view an Event-B machine M as a labelled transition system with state space S and event labels E. The event labels are as just outlined above. The states are given by the possible values of the machine variables. We view the events as defining a labelled transition relation on the state space of the following type:

$$A \in E \rightarrow (S \leftrightarrow S)$$

That is for event label $ev.i \in E$, the event ev with parameters instantiated to i defines a transition relation on the state space, $A(ev.i) \in S \leftrightarrow S$. We can lift A to traces of event labels giving $\bar{A} \in seq(E) \rightarrow (S \leftrightarrow S)$. For an event trace t, $\bar{A}(t)$ represents the relational composition of the transition relations of each event label in t. This is defined inductively over traces as follows:

$$\begin{aligned} \bar{A}(\ \langle\rangle\ ) &= ID \\ \bar{A}(\ \langle e \rangle t\ ) &= A(e); \bar{A}(t) \end{aligned}$$

Here ID is the identify relation on states ($skip$). If I is the set of initial states of a machine M, then $\bar{A}(t)[I]$ is the set of states reachable by executing trace t. We say that t is a trace of M if the set of states reachable by executing trace t is non-empty:

$$t \in traces(M) \quad \text{iff} \quad \bar{A}(t)[I] \neq \varnothing$$

Note that traces are prefix-closed by this definition, that is, if t is a trace then any prefix of t is also a trace.

In the previous section we saw how the Inc event could be treated as hidden in order to relate the traces of the refined machine with the traces of the abstract machine. Since we don't care about the identity of an event when it is hidden, for convenience of definition we can asume that all hidden events are combined into a single unlabelled transition relation H:

$$H \in S \leftrightarrow S$$

The definition of the lifted transition relation on event sequences is then modified to take account of H as follows:

$$\begin{aligned} \bar{A}(\ \langle\rangle\ ) &= H^* \\ \bar{A}(\ \langle e \rangle t\ ) &= H^*; A(e); \bar{A}(t) \end{aligned}$$

Now we give a definition to refinement in terms of trace inclusion, that is, machine $M1$ is refined by $M2$ when $traces(M2) \subseteq traces(M1)$. That is, any possible behaviour of $M2$ is a possible behaviour of $M1$.

New events may be introduced in Event-B refinement, that is, a refined machine may have additional events that have no corresponding events in the abstract machine. New events are required to refine $skip$. This ensures that they have no effect in terms of the abstract state. 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.

The proof obligations defined for Event-B refinement are based on the following proof rule that makes use of a gluing invariant J:

- Each $M1.A$ is (data) refined by $M2.A$ under J
- Each $M2.H$ refines skip under J

It can be shown that these are sufficient conditions for trace refinement [8]. In the simple concurrent program of Section 2 we saw how the gluing invariant $x = card(Inc)$ was used to discharge the conditions showing that the abstract Out event is refined by the concrete Out event and that the Inc event in the refined model refines $skip$.

4. 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 $skip$.

We have already seen an example of this in our Event-B treatment of the simple concurrent program in Section 2. The abstract model consists of a single event that outputs the value N. We view the refined model as breaking the atomicity of the output event by introducing the Inc event that models the behavior of the parallel sub-processes. The decomposition of the atomicity of the simple concurrent program is modelled diagrammatically in Figure 5. This diagrammatic notation is based on JSD (Jackson Structure Diagrams) by Jackson [14]. Figure 5 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 p. An important feature of this diagrammatic notation, in common with JSD 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 as discussed previously) and then executing $Out(x)$.

Another important feature of the diagrammatic notation 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 $Out(N)$. In Section 2 the proof obligations that we discharged were concerned with proving precisely these refinement conditions.

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

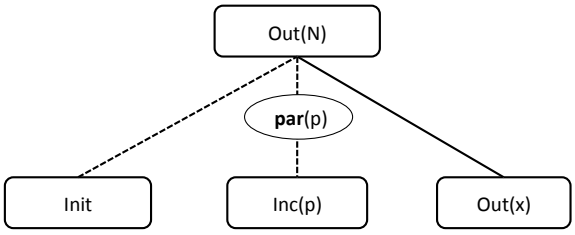


Figure 5. Illustration of the atomicity decomposition

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 files to their contents. The $Write$ event has 2 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 content is structured as a set of pages of data so that the type $CONT$ is defined as follows:

$$CONT = PAGE \nrightarrow DATA$$

Figure 6 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)). The occurrence of the event $PageWrite(f, p)$ models writing of page p of file f.

In order to model the event sequencing implied by Figure 6, we introduce variables corresponding to the $StartWrite$ and $PageWrite$ events as follows:

Invariants

$inv1 : StartWrite \subseteq FILE$
$inv2 : PageWrite \subseteq FILE \times PAGE$

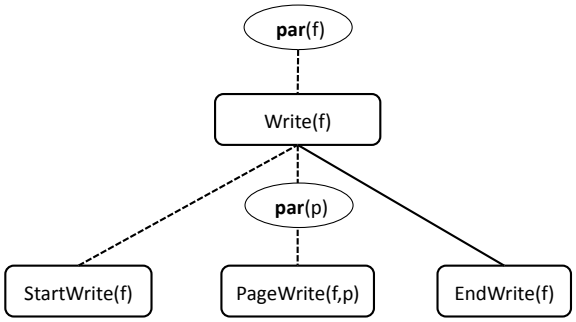


Figure 6. Decomposition of the atomiticy of file write

The types of these variables are determined by the parallel quantification in Figure 6. 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:

$inv3: writebuf \in StartWrite \rightarrow CONT$
$inv4: sdsk \in StartWrite \rightarrow CONT$

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:

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
$act1: PageWrite := PageWrite \cup \{f \mapsto p\}$
$act2: sdsk(f) := sdsk(f) \lhd\!\!+ \{p \mapsto d\}$
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 Figure 6). 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 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):

$\langle\ StartWrite.f1.c1,\ \ PageWrite.f1.p1.c1(p1),$

$StartWrite.f2.c2,\ \ PageWrite.f1.p2.c1(p2),$

$PageWrite.f2.p1.c2(p1),\ \ PageWrite.f2.p2.c2(p2),$

$\mathbf{EndWrite.f2.c2},\ \ PageWrite.f1.p3.c1(p3),\ \ \mathbf{EndWrite.f1.c1}\ \rangle$

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 new the 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 fomal connection. However, the diagram of Figure 6 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.

5. 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.

Parallel Composition of Machines

In general, an event has the form

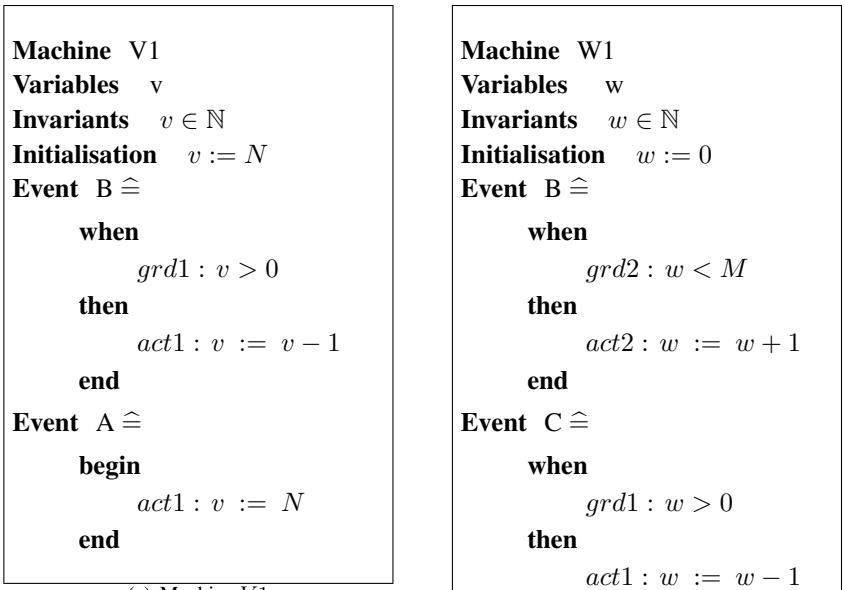
any x **where** G **then** S **end**

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:

$$ev1 \quad = \quad \textbf{any } y \textbf{ where } G(y,m) \textbf{ then } S(y,m) \textbf{ end}$$

$$ev2 \quad = \quad \textbf{any } z \textbf{ where } H(z,n) \textbf{ then } T(z,n) \textbf{ end}$$



(a) Machine $V1$

(b) Machine $W1$

Figure 7. Machines to be composed in parallel

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

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 *V1* and *W1* of Figure 7. 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 Figure 8. The A event and C event of *VW1* come directly from *V1* and *W1* respectively as they are not joint events (i.e., independent events). The B event is a joint event and is defined as the fusion of the B-events of *V1* and *W2*. The initialisations of *V1* and *W1* are also combined to form the initialisation of *VW1*. The joint B event simultaneously decreases v while increasing w, provided $v > 0$ and $w < N$.

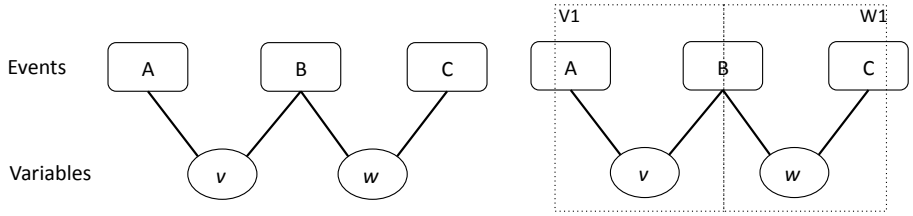
Machine VW1
Variables v, w
Invariants $v \in \mathbb{N},\ w \in \mathbb{N}$
Initialisation $v := N,\ w := 0$
Event A $\widehat{=}$
begin
$act1 : v := N$
end
Event B $\widehat{=}$
when
$grd1 : v > 0$
$grd2 : w < M$
then
$act1 : v := v - 1$
$act2 : w := w + 1$
end
Event C $\widehat{=}$
when
$grd1 : w > 0$
then
$act1 : w := w - 1$
end

Figure 8. Composition of $V1$ and $V2$.

Decomposition

We have presented $VW1$ as having been formed from the composition of $V1$ and $W1$. We can view the relationship between these machines in another way. Let us suppose we had started with $VW1$ and decided that we wish to decompose it into subsystems. The diagram in Figure 9(a) illustrates the dependencies between events and variables in the machine $VW1$. For example, the line from the box indicating event A to the circle 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 Figure 9(b) where variables v and w of $VW1$ are partitioned into subsystems $V1$ and $W1$ respectively, A is an event of subsystem $V1$, C is an event of subsystem $W1$ and B is an event shared by both subsystems.

Event B of system $VW1$ is partitioned into two parts, one of which will belong in $W1$ and the other in $W1$. Event B 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.



(a) Variable access by events in VW (b) Split events and variables

Figure 9. Illustration of decomposition a machine

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 $VW1$ into the separate B events of $V1$ and $W1$ respectively.

Fusion with shared event parameters

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:

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

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

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:

Event B $\widehat{=}$

any i

where

$grd1 : 0 \leq i \leq v$

$grd2 : w < N$

then

$act1 : v := v - i$

$act2 : w := w + i$

end

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.

Independent refinement of subsystems

In Section 3, we saw how the event traces of a machine may be defined and it was pointed out that such event traces are used to define the behaviour of CSP processes as found in [13]. Parallel composition of processes in CSP is defined by processes synchronising over shared events and non-shared events can occur independently in sub-processes. The semantics of parallel composition in CSP is defined in terms of an operator over traces that fuses shared events and interleaves non-shared events. Details may be found in [13]. The parallel composition of Event-B machines that we use here achieves the same effect as CSP parallel composition by synchronising shared events and leaving non-shared events independent. For example, in composing $V1$ and $W1$, the shared B events are synchronised by event fusion while the non-shared events, A and C, remain independent in the composition. It can be shown that this composition of machines results in the same composition of event traces as found in the CSP definition. Details may be found in [7].

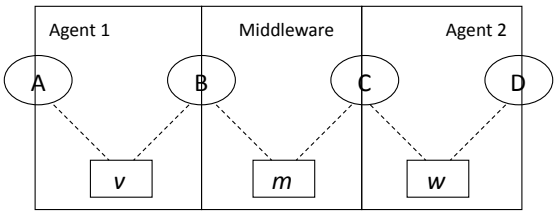


Figure 10. Decomposition with asynchronous middleware

Significantly parallel composition of event traces is monotonic w.r.t. trace refinement [13]. Therefore a corollary of the correspondence between CSP parallel composition and our composition of Event-B machines is that our composition is also monotonic w.r.t. trace refinement. This means that 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.

6. Incremental development of a distributed file transfer

In this section we present an incremental development of a simple system for copying a file from one location to another. 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 Figure 10. 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.

6.1. Abstract model

The model makes use of the following context which introduces the types $PAGE$ and $DATA$ respectively. A file is modelled as a partial function from pages to data. The constant file $f0$ will be used in the initialisation of the machine.

Context C1
Sets *PAGE* ; *DATA*
Constants $f0$
Axioms

$axm1: \ f0 \in PAGE \pfun DATA$

END

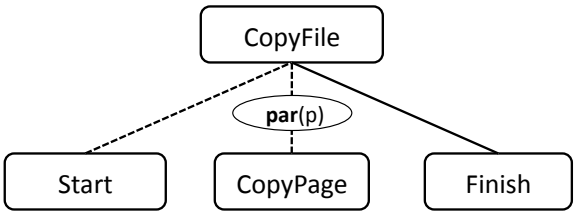


Figure 11. Refining atomicity of the $CopyFile$ event

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 variables are initialised as follows:

Initialisation

$act1: \ fileA := f0$
$act2: \ fileB := \varnothing$

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

6.2. Breaking atomicity

The atomicity of the $CopyFile$ event is decomposed in the same way in which the atomicity of the $Write$ event was decomposed in Section 4. This is illustrated in Figure 11. We introduce control variables based on this diagram as well as a buffer buf in which pages are written one at a time:

Machine F2
Refines F1
Variables *fileA* , *fileB* , *Start* , *CopyPage* , *Finish* , *buf*
Invariants

$inv1: \ Start \in BOOL$ Control variable
$inv2: \ CopyPage \subseteq PAGE$ Control variable
$inv3: \ Finish \in BOOL$ Control variable

$inv4 : buf \in PAGE \pfun DATA$ Page buffer

The control behaviour of the $Start$ and $CopyPage$ events is constructed based on the control implied by the diagram in Figure 11 (in the same way that the control for the file write process was defined in Section 4). In addition the $CopyPage$ event writes a page from $fileA$ to buf. We omit the definitions of these events and focus on the $Finish$ event that refines the abstract $CopyFile$ event. The $Finish$ event is enabled once all pages have been copied into buf, i.e., $card(buf) = card(fileA)$. The $Finish$ event copies buf into $fileB$ and sets the control variable $Finish$ to $TRUE$:

Event Finish $\widehat{=}$

Refines CopyFile

when

$grd1 : Start = TRUE$

$grd2 : Finish = FALSE$

$grd3 : card(buf) = card(fileA)$

then

$act1 : fileB := buf$

$act2 : Finish := TRUE$

end

END

The $Finish$ event gives rise to an unproved proof obligation as follows:

$$Start = TRUE,\ \ card(buf) = card(fileA)$$
$$\vdash$$
$$buf = fileA$$

This proof obligation embodies the need to demonstrate that the effect of the $Finish$ event (which assigns buf to $fileB$) refines the effect of the $CopyFile$ event (which assigns $fileA$ to $fileB$). The requirement is to show that the value assigned to $fileA$ is the same in the abstract and refined events hence the goal $buf = fileA$. Clearly while we are copying pages from $fileA$ to buf it is not always the case that $buf = fileA$. But we would expect that any page-content pair that has been copied into buf is also a pair of $fileA$, i.e., that $buf \subseteq fileA$. Thus we add the following invariant to the model:

$$inv5 : Start = TRUE \Rightarrow buf \subseteq fileA$$

With this invariant available as an additional hypothesis, the above proof obligation can be discharged since $buf \subseteq fileA$ and $card(buf) = card(fileA)$ together mean that $buf = fileA$.

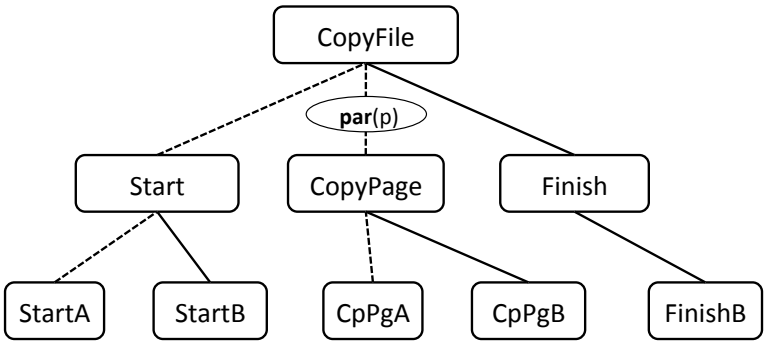


Figure 12. Splitting events into sender and receiver parts

6.3. Split events to A side and B side

As previously mentioned, we will decompose the file transfer system into three subsystems. We 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 Figure 12 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.

We replace the buf variable from the previous level by two variables $bufA$ and $bufB$ representing the values of the buffers in the A and B sides respectively. We also introduce variables $sizeA$ and $sizeB$ representing the size of $fileA$ as known at sides A and B respectively. As indicated by Figure 12, the $StartA$ event refines $skip$ and is defined as follows:

Event StartA $\widehat{=}$

when

$grd1 : StartA = FALSE$

then

$act1 : StartA := TRUE$

$act2 : bufA := \varnothing$

$act3 : sizeA := card(fileA)$

end

As indicated by Figure 12, the $StartB$ event refines the $Start$ event of the previous level. The receiving side will discover the file size when it starts the transfer so the $StartB$ event sets the $sizeB$ variable appropriately.

Event StartB $\widehat{=}$
Refines Start

when

$grd1 : StartB = FALSE$
$grd2 : StartA = TRUE$

then

$act1 : StartB := TRUE$
$act2 : bufB := \varnothing$
$act3 : sizeB := sizeA$

end

We omit details of the $CopyPageA$ and $CopyPageB$ events.

The $FinishB$ event refines the $Finish$ event of the previous level (with control variables related by invariant $FinishB = Finish$). It is enabled when the number of pages received reaches the number expected ($grd3$):

Event FinishB $\widehat{=}$
Refines Finish

when

$grd1 : FinishB = FALSE$
$grd2 : StartB = TRUE$
$grd3 : card(bufB) = sizeB$

then

$act1 : fileB := buf$
$act2 : FinishB := TRUE$

end

END

6.4. Introduce message variables

Now consider again the $StartB$ event just presented. 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: $grd2$ refers to the $StartA$ variable and $act3$ refers to the $sizeA$ 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 ($StartM$, $CopyPageM$, $sizeM$ and $bufM$). These duplicate variables will be separated into a middleware machine (Figure 10) and become abstract representations of messages in transit in the middleware.

We refine the $StartA$ event so that it initialises the duplicate variables as well as the sender variables:

Event StartA $\widehat{=}$

Refines StartA

when

$grd1 : StartA = FALSE$

then

$act1 : StartA := TRUE$

$act2 : bufA := \varnothing$

$act3 : sizeA := card(fileA)$

$act4 : StartM := TRUE$

$act5 : bufM := \varnothing$

$act6 : sizeM := card(fileA)$

end

Now, instead of being enabled when the $StartA$ flag is true, the $StartB$ event is enabled when the $StartM$ flag is true. Also the value assigned to $sizeB$ is based on $sizeM$ rather than $sizeA$ as previously:

Event StartB $\widehat{=}$
Refines StartB

when

$grd1 : StartB = FALSE$

$grd2 : StartM = TRUE$

then

$act1 : StartB := TRUE$

$act2 : bufB := \varnothing$

$act4 : sizeB := sizeM$

end

This refinement relies on the invariants $StartM = StartA$ and $sizeM = sizeA$.

6.5. Separate machines

The previous model is decomposed into three separate machines representing three subsystems as illustrated in Figure 10. 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.

Figure 13 provides an architectural overview of the decomposition illustrating how the variables and events are distributed amongst the subsystems. The variables allocated

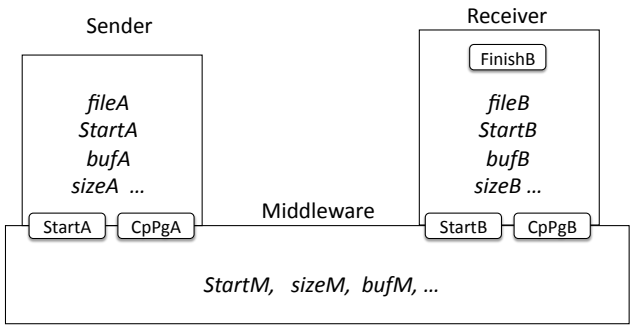


Figure 13. Architectural illustration of decomposition

to each subsystem are listed in italic in the relevant box for that 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.

Let us look at how the $StartA$ event is split into the sender event and the middleware event. We would like guard $grd1$ to become a guard of the sender event. We would like actions $act1..act3$ to become actions of the sender event and actions $act4...act6$ to become actions of the middleware event. However, this is not possible since $act6$ refers to $card(fileA)$. We cannot have an event of the middleware referring to a variable of the sender. We overcome this by adding a parameter s to the event whose value is the same as $card(fileA)$ and use this in the action $act6$:

Event StartA $\widehat{=}$
Refines StartA

any s
where
$grd1 : StartA = FALSE$
$grd2 : s = card(fileA)$
then
$act1 : StartA := TRUE$
$act2 : bufA := \varnothing$
$act3 : sizeA := card(fileA)$
$act4 : StartM := TRUE$
$act5 : bufM := \varnothing$
$act6 : sizeM := s$
end

The parameter s that we have just introduced will become a shared parameter when the $StartA$ event is split into separate sender and middleware events.

We are now in a position to partition the $StartA$ event into the sender event $mA1 : StartA$ and the middleware event $mM1 : StartA$.

Event mA1:StartA $\widehat{=}$

any s
where
$grd1 : StartA = FALSE$
$grd2 : s = card(fileA)$
then
$act1 : StartA := TRUE$
$act2 : bufA := \varnothing$
$act3 : sizeA := card(fileA)$
end

Event mM1:StartA $\widehat{=}$

any s
where
$grd1 : s \in \mathbb{N}$
then
$act1 : StartM := TRUE$
$act2 : bufM := \varnothing$
$act3 : sizeM := s$
end

6.6. Introducing message types

In this section we show how more explicit datatypes representing messages can be introduced and used in the file transfer middleware. We introduce a type representing general message structures $MESS$ and a type MID representing message identifiers associated with messages. We regard elements of $MESS$ as structured messages, each of which has an identifier field. This identifier field is represented by a projection function idF mapping $MESS$ to MID:

$axm1 : idF \in MESS \rightarrow MID$

We define a subtype of $MESS$ called $StartMESS$ representing the messages that can be sent from sender to receiver to start a file transfer. Each $StartMESS$ has a size field $sizeF$ which is used to indicate the size of the file to be transferred.

$axm2 : StartMESS \subseteq MESS$
$axm3 : sizeF \in StartMESS \rightarrow \mathbb{N}$

Since $StartMESS$ is a subset of $MESS$, each $StartMESS$ will also have an idF field.

We introduce another subtype of $MESS$ called $PageMESS$ representing messages that are used to transfer pages from sender to receiver. Each $PageMESS$ has a page field and a data field.

$axm4 : PageMESS \subseteq MESS$
$axm5 : pageF \in PageMESS \rightarrow PAGE$
$axm6 : dataF \in PageMESS \rightarrow DATA$

We assume that the two subtypes are distinct:

$axm7 : StartMESS \cap PageMESS = \varnothing$

We refine the communications medium by replacing $StartM$ with a set of $StartMESS$ messages and replace $bufM$ with a set of $PageMESS$ messages. It also contains a variable mid representing the set of message identifiers already used. This allows for specification of a freshness constraint on the choice of message identifier.

Machine mM2
Refines mM1
Variables *StartMS* , *CopyPageMS* , *mid*
Invariants

$inv1 : StartMS \subseteq StartMESS$
$inv2 : CopyPageMS \subseteq PageMESS$
$inv3 : mid \subseteq MID$

The relationship between the abstract sets of the previous model of the communications medium and the message sets that replace them is given by the following gluing invariants:

$inv4 : StartMS \neq \varnothing \ \Rightarrow \ StartM = TRUE$
$inv5 : \forall m \cdot m \in StartMS \ \Rightarrow \ sizeF(m) = sizeM$
$inv6 : \forall m \cdot m \in CopyPageMS \ \Rightarrow \ pageF(m) \in CopyPageM$
$inv7 : \forall m \cdot m \in CopyPageMS \ \Rightarrow \ pageF(m) \mapsto dataF(m) \in bufM$

Now the $StartA$ event of the middleware is parameterised by a message m whose idF field is fresh. The relationship between the parameter m and the abstract parameter s is given by the witness statement which specifies that the size parameter s equals the value of the size field of m. The selected message identifier is added to the set mid indicating that it should not be used again.

Event mM2:StartA $\widehat{=}$
Refines mM1:StartA

any m
where
$grd1 : m \in StartMESS$
$grd2 : idF(m) \notin mid$
witness
$s : s = sizeF(m)$
then
$act1 : StartMS := StartMS \cup \{m\}$
$act2 : mid := mid \cup \{idF(m)\}$
end

The previous $StartB$ event of the middleware is enabled when $StartM = TRUE$. This is refined by there being a message m in $initMS$:

Event mM2:StartB $\widehat{=}$
Refines mM1:StartB

any m
where
$grd2 : m \in StartMS$
witness
$s : s = sizeF(m)$
end

The $CopyPageA$ and $CopyPageB$ events are refined in a similar way. Details are omitted.

7. Concluding

Our initial exploration of JSD structure diagrams as a means of representing the structure of atomicity decomposition was influenced by the work of Ball [6] on the use of KAOS [10] 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.

It was shown how Event-B can be use to model and reason about distributed systems from a high-level global view down to a detailed distributed architectural view. It was shown how refinement and decomposition can be used to introduce distribution of state and control and to introduce message passing between components.

The key ideas that we covered include concurrency and atomicity in Event-B models, refining the atomicity of events in refinement and decomposing models into submodels based on synchronous interaction between subsystems. In this context we treat distributed systems as a special case of concurrency where the only shared variables are buffers used for message-passing. This is very convenient as it allows us to reason about key properties of systems using simpler global abstractions of state and then refine these to distributed systems.

This incremental style of development has been applied to many distributed systems including a replicated database [16] and an electronics funds transfer system [9].

Acknowledgements

Some of 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. *Modelling in Event-B: System and Software Engineering*. To be published by Cambridge University Press, 2008.

[2] J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *ICFEM 2006 Lecture Notes in Computer Science*, volume 4260, 2006.

[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] R.J.R. Back and R. Kurki-Suonio. Decentralisation of process nets with centralised control. In *2nd ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*, pages 131–142, 1983.

[5] R.J.R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Sci. Comp. Prog.*, 13:133–180, 1989.

[6] 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.

[7] M.J. Butler. *A CSP Approach To Action Systems*. D.Phil. Thesis, Programming Research Group, Oxford University, 1992. `http://eprints.ecs.soton.ac.uk/974/`.

[8] M.J. Butler. Stepwise refinement of communicating systems. *Science of Computer Programming*, 27(2):139–173, September 1996.

[9] M.J. Butler and D.S. Yadav. An incremental development of the mondex system in event-b. *Formal Aspects of Computing*, 20(1):61–77, January 2008.

[10] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Sci. Comput. Program.*, 20(1-2):3–50, 1993.

[11] D. Gries. *The Science of Programming*. Springer–Velag, 1981.

[12] S. Hallerstede. On the purpose of event-b proof obligations. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *ABZ*, volume 5238 of *Lecture Notes in Computer Science*, pages 125–138. Springer, 2008.

[13] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice–Hall, 1985.

[14] M.A. Jackson. *System Development*. Prentice–Hall, 1983.

[15] C.C. Morgan. Of wp and CSP. In W.H.J. Feijen, A.J.M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer–Verlag, 1990.

[16] 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.