Shared Event Composition/Decomposition in Event-B

Renato Silva and Michael Butler

School of Electronics and Computer Science
University of Southampton, UK
{ras07r,mjb}@ecs.soton.ac.uk

Abstract. The construction of specifications is often a combination of smaller sub-components. *Composition* and *decomposition* are techniques that support reuse and allow us to formally combine sub-components through refinement steps while reusing their properties. Sub-components can result from a design or architectural goal and a refinement framework should allow further parallel development over the sub-components. We propose the definition of composition and decomposition in the Event-B formalism following a shared event approach where sub-components interact via synchronisation over shared events and shared states are not allow. We define the necessary proof obligations to ensure a valid composition or decomposition. We also show that shared event composition preserves refinement proofs for sub-components, that is, in order to maintain refinement of compositions, it is sufficient to prove refinement between corresponding subcomponents. A case study applying these two techniques is illustrated using Rodin, the Event-B toolset.

Key words: formal methods, composition, decomposition, reuse, Event-B, design techniques, specification

1 Introduction

The development of specifications in a "top-down" style starts with an abstract model of the envisaged system. Systems can often be seen as a combination and interaction of several sub-specifications (hereafter called sub-components) where each sub-component has its own functionality aspect. This view introduces *modularity* in the system: different sub-components represent a particular functionality and changes in the sub-components are accommodated more gracefully [1] in the system specification. We use *composition* to structure specifications through the interaction of sub-components seen as independent modules. This use of composition is not new in other formal notations: examples are [2,3,4]. Here we express how we can use (and reuse) composition for building specifications in Event-B [5] through sub-components (modules) interaction, benefiting from their properties and proof obligations (POs). The interesting part of composition involves the interaction of sub-components which usually occurs by shared state [6], shared operations [7] or a combination of both (for example, fusion composition [4]). Although sub-components have states, we mainly focus on their (visible) events similar to CSP [8,9]: we follow a *shared event composition approach* where events are synchronised in parallel. Decomposition is motivated by the possibility of breaking a complex problem or system into parts that are easier to conceive, manage and maintain. The partition of a model into sub-components can also be seen as a design/architectural decision and the further development of the sub-components in parallel is possible. Besides alleviating the complexity for large systems and respective proofs, decomposition allows team development in parallel over the same model which is very attractive in the industrial environment. Moreover the proof obligations of the original (non-decomposed) model can be reused by the sub-components. We present in more detail the shared event approach applied to composition and decomposition. Moreover, the proof obligations to ensure a valid composition are expressed including the possibility to reuse the sub-components properties. The monotonicity property for composition is proved by means of refinement proof obligations. We see decomposition as the inverse operation of composition and therefore we can reuse its properties to decompose systems. A guideline in how to apply a shared event decomposition is presented and a case study is illustrated to highlight the use of this technique. The models are developed in the Rodin [10], which is a toolset for Event-B[5,11].

This document is structured as follows: Section 2 gives an overview of the Event-B formal method. Section 3 introduces the notion and motivation for shared event approach for composition and decomposition. Composed machines, properties, proof obligations are described in Sect. 4. A guideline in how to use decomposition is presented in Sect. 5. Section 6 illustrates the application of composition and decomposition to a distributed system case study: file access system. Related work, conclusions and future work are drawn in Section 7.

2 Event-B Language

Event-B is a formal methodology that uses mathematical techniques based on set theory and first order logic supporting system development with abstract specification. An abstract Event-B specification is divided into a static part called *context* and a dynamic part called *machine*. A machine *sees* as many contexts as desired. A context consists of sets s (collection of elements or a type definition), constants c and axioms $A(\dots)$ of the system. A machine contains the state (global) variables v whose values are assigned in *events*. Events that can be parameterised (local variables p) occur when enabled by their *guards* $G(\dots)$ being true and as a result *actions* $S(\dots)$ are executed. *Invariants* $I(\dots)$ define the dynamic properties of the specification and POs are generated to verify that these properties are always maintained. An event *evt* is expressed by parameters p, by guards $G(s,c,p,v)$ and actions $S(s,c,p,v,v')$:

$$evt \mathrel{\widehat{=}} \textbf{ANY } p \textbf{ WHERE } G(s,c,p,v) \textbf{ THEN } S(s,c,p,v,v') \textbf{ END}.$$

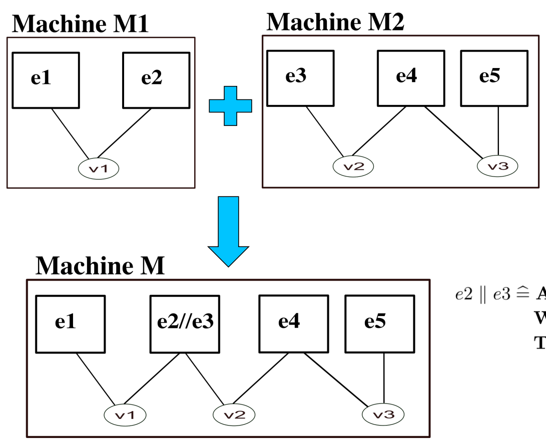
When the guard $G(s,c,p,v)$ is true then the event *evt* is enabled and therefore the action $S(s,c,p,v,v')$ updates the set of variables v to v' (value of v after the assignment). An abstract Event-B specification can be refined with the introduction of more details and becoming closer to a concrete implementation. A context *extends* an abstract context by adding sets, constants or axioms. The abstract context properties are still assumed. Refinement of a machine consists of refining existing events. The relation between variables in the concrete and abstract model is given by a *gluing invariant* $J(\dots)$. POs are generated to ensure that this invariant is preserved in the concrete model. New events can be added, refinining *skip* which may be declared as convergent, meaning they do not cause divergence. The convergence is proved if each new event decreases a *variant*. The variant must be well-founded and may be an integer or a finite set.

3 Shared Event Approach

The shared event approach seems suitable for the development of distributed systems[7]: sub-components interact through synchronised events in parallel; moreover sub-components can communicate using shared parameters which is useful for modelling message broadcasting systems.

3.1 Shared Event Composition

Sub-component specifications that are part of a full system specification, deal with a particular part of the system being modelled. Sub-component interaction must be verified to comply with the desired behavioural semantic of the system. Here we focus on the developments using shared event composition where individual elements' properties are conjoined: *conjunction* of individual invariants, *conjoining* variables and *synchronisation* of events.



$e2 \mathrel{\widehat{=}} \textbf{ANY } p$
$\quad \textbf{WHERE } p \in \mathbb{N} \wedge v1 = 0$
$\quad \textbf{THEN } v1 := p$

$e3 \mathrel{\widehat{=}} \textbf{ANY } p$
$\quad \textbf{WHERE } v2 = p$
$\quad \textbf{THEN } v2 := 0$

$e2 \parallel e3 \mathrel{\widehat{=}} \textbf{ANY } p$
$\quad \textbf{WHERE } p \in \mathbb{N} \wedge v1 = 0 \wedge v2 = p$
$\quad \textbf{THEN } v2 := 0 \parallel v1 := p$

Fig. 1. Shared event composition of *M1* and *M2* (a) resulting in *M* (b)

Consider Fig. 1 where machine *M1* has events *e1* and *e2* that use variable *v1*. Moreover machine *M2* has events *e3*, *e4* and *e5* using variables *v2* and *v3*. If events *e2* and *e3* occur in parallel, they can be synchronised: machines *M1* and *M2* are composed by *sharing events*. In Fig. 1, machine *M* is the result of the composition of machines *M1* and *M2* where *e2* from machine *M1* and *e3* from machine *M3* are composed: *e2* ‖ *e3*. The interaction of machines *M1* and *M2* through their events results in a *composed event* sharing two independent variables: *v1* and *v2*. A general definition for the parallel composition of events $e2$ and $e3$ is defined as Def. 1 [7]:

Definition 1. *Composition of events* $e2$ *and* $e3$ *with parameter* p *results in:*

$$\begin{aligned} e2 &\mathrel{\widehat{=}} \textbf{\textit{ANY}}\ p?, x\ \ \textbf{\textit{WHERE}}\ p? \in C \wedge G(p?, x, m)\ \ \textbf{\textit{THEN}}\ S(p?, x, m)\ \ \textbf{\textit{END}} \\ e3 &\mathrel{\widehat{=}} \textbf{\textit{ANY}}\ p!, y\ \ \textbf{\textit{WHERE}}\ H(p!, y, n)\ \ \textbf{\textit{THEN}}\ T(p!, y, n)\ \ \textbf{\textit{END}} \\ e2 \parallel e3 &\mathrel{\widehat{=}} \textbf{\textit{ANY}}\ p!, x, y\ \ \textbf{\textit{WHERE}}\ p! \in C \wedge G(p!, x, m)\ \wedge\ H(p!, y, n) \\ &\qquad \textbf{\textit{THEN}}\ S(p!, x, m) \parallel T(p!, y, n)\ \ \textbf{\textit{END}} \end{aligned}$$

where x, y, p are sets of parameters from each of the events $evt1$ and $evt2$. Event $evt1$ has $p?$ as an input parameter and $evt2$ has $p!$ as an output parameter and the resulting composition is $p!$ itself an output parameter (like in CSP). This property can be used to model message-passing systems: $e3$ sends a message to $e2$ using the parameter p. Communication between input type parameters is also possible but not with both output parameters since this could lead to a deadlock state [7].

Action systems [12] provides a general description of reactive systems, capable of modelling terminating, aborting and infinitely repeating systems. Event-B is inspired in action systems and can be seen as a realisation of actions systems but using a combination of logic and mathematics as a formal language. Both formalisms share the same refinement semantics. Therefore we claim that Event-B has the same semantic structure and refinement definitions as action systems. It is possible to make a correspondence between parallel composition in CSP and an event-based view of parallel composition for action systems [13,14].

Theorem 1. *The shared event parallel composition of actions systems corresponds to the CSP parallel-composition. The failure-divergence semantics of CSP can be applied to action systems. The failure divergence semantics of action system M in parallel with N,* $M \parallel N$ *is defined as:*

$$[\![M \parallel N]\!] = [\![M]\!] \parallel [\![N]\!]$$

where $[\![M]\!]$ *and* $[\![N]\!]$ *are the failure divergence semantics of* M *and* N *respectively. The proof of this theorem can be found in [13].*

The semantics of the parallel composition of action systems M and N corresponds to the set of failure-divergence for each individual action system in parallel. From the correspondence between action systems and Event-B, M and N can be refined independently which is one of the most important and powerful properties that shared event composition in Event-B inherits from CSP. The monotonicity property for the shared event composition in Event-B is proved by means of proof obligation in Sect. 4.3.

When sub-components are composed it is desirable to define properties that relate the individual sub-components allowing interactions. These properties are expressed by adding *composition invariants* $I_{CM}(s, c, v1, \ldots, vm)$ to the composed machine constraining the variables of all machines being composed.

Definition 2. *The invariant of the parallel composition of machines* $M1$ *to* Mn *with variables* $v1$ *to* vn *respectively is the conjunction of the individual invariants and the composition invariant* $I_{CM}(s, c, v1, \ldots, vn)$*:*

$$I(M1 \parallel \cdots \parallel Mm) \mathrel{\widehat{=}} I_1(s, c, v1) \wedge \cdots \wedge I_m(s, c, vm) \wedge I_{CM}(s, c, v1, \ldots, vn). \qquad (1)$$

In Fig. 1, *composed machine M* has as invariant the conjunction of the individual invariants $I(A \parallel B) \mathrel{\widehat{=}} I_A(s, c, v1) \wedge I_B(s, c, v2, v3)$ plus possible composition invariant $I_{CM}(s, c, v1, v2, v3)$. In a shared event composition the sub-components have independent state space (variables are unique to each machine). Consequently composition reasoning is simplified as there are no constraints between state spaces of sub-components.

3.2 Shared Event Decomposition

Decomposition can be seen as the inverse process of composition: after some refinements a larger model may be decomposed into smaller components. This step might be a consequence of complexity or just as an architectural decision. The shared event approach is also used: events are shared between sub-components and variable sharing is not allowed. Butler [15] proposes a shared event decomposition for Event-B inspired by CSP and action systems with event sharing as seen in Fig. 2. We follow that work in our approach. Events using variables allocated to different sub-components (*e2* shares *v1* and *v2*) must be split. The part corresponding to each variable (*e2'* and *e2"*) is used to create partial versions of the original event. After the decomposition, the individual machines can be further refined since the composition relation holds. The possible recomposition of the sub-components (or their refinements) is a refinement of the original composed component although this step should never be required in practice. Figure 3 shows the decomposition of *M1* into *M3_0* and *M4_0* that are further refined into *M3_m* and *M4_n* respectively. At this stage a possible recomposition of *M3_m* and *M4_n* into *cM2* should be proved to be a refinement of *M1*.

4 Composed Machines: Composition and Refinement

We define a new construct *composed machine*, representing the shared event composition of Event-B machines. We aim to have a construct that remains reactive to changes in the sub-components. Consequently the composition is *structural*. The interaction of sub-components follows a "top-down" approach, representing a *refinement* of an existing abstraction. To formalise the composition, it is necessary to define composition and refinement POs. In the following sections, we

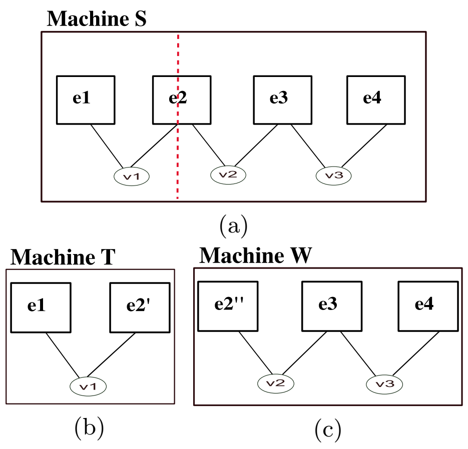


Fig. 2. Shared Event Decomposition of Machine S in Machines T and W with shared event $e2$

introduce the structure of a composed machine, respective POs and prove the monotonicity property.

4.1 Structure of Composed Machines

A shared event composed machine is expressed as the parallel conjunction of sub-component properties. Machines are composed in parallel including their properties and events: $CM \mathrel{\widehat{=}} M1 \parallel \cdots \parallel Mm$ as seen in Fig. 4. Moreover:

- The composed machine variables are all the sub-component variables (v_1 from $M1$, v_2 from $M2$, ..., v_m from Mm) and are state-space disjoint.
- The invariants of the composed machine are defined as Def. 2.
- The composed events are defined according to Def. 1.

When a composed machine is used as a combination of composition and refinement, it refines an abstract model and just like in an ordinary machine, abstract events must be refined. For instance, a composed machine CM resulting from the parallel composition of $M1 \ldots Mm$ and refining abstract machine $M0$ can be expressed as $M0 \sqsubseteq CM \equiv M0 \sqsubseteq M1 \parallel \cdots \parallel Mm$. Next we present the required POs to verify composed machines.

4.2 Proof Obligations

POs play an important role in Event-B developments. POs are generated to verify the properties of a model. For simplicity we define POs in terms of a composition of two machines M_1 and M_2 that refine machine M_0, but the rules generalise easily to the composition of n machines. Furthermore context elements in the formulas $(s, c, A(s, c))$ are not considered. The POs defined for standard machines are [5]:

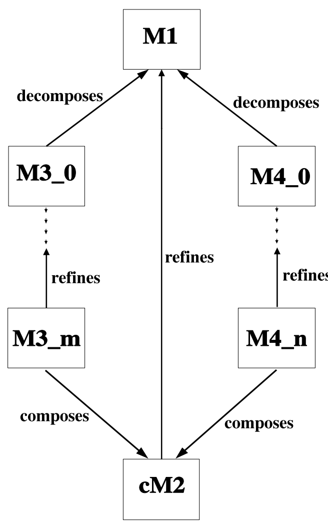


Fig. 3. Decomposition, Recomposition and Refinement

COMPOSED MACHINE CM
INCLUDES $M_1, \ldots, M_m$
VARIABLES $v_1, \ldots, v_m$
INVARIANTS $I_{CM}(s, c, v_1, v_2, \ldots, v_m)$
EVENTS
$evt_{11} \mathrel{\widehat{=}} M1.evt_{11} \parallel \ldots Mm.evt_{m1}$
$\ldots$
$evt_{1p} \mathrel{\widehat{=}} M1.evt_{1p} \parallel \ldots Mm.evt_{m1}$
evt_{1p}
END

Fig. 4. Composed machine CM composing machines $M1$ to Mm seeing context Ctx

- Consistency: Invariant Preservation (INV) and Feasibility (FIS)
- Refinement: Guard Strengthening (GRD), Simulation/Refinement (SIM) and Gluing Invariant Preservation (INV)
- Variant: Numeric Variant (NAT), Numeric Variant Decreasing (VAR), Finite Set Variant (FIN)
- Well-Definedness(WD)

These POs also are defined for composed machines except the ones related with variant (no variant for composed machines). We simplify the composed machines POs by assuming that the POs of the individual machines hold. We define the additional POs necessary to ensure that the composed machine satisfies all the standard POs. We consider that the POs of the $M0$, M_1 and M_2 hold. The respective composition POs are described as follows.

Consistency Consistency POs are required to be always verified. The feasibility proof obligation for the composed event $evt1 \parallel evt2$ is $FIS_{evt1 \parallel evt2}$.

Theorem 2. *The individual FIS PO for each event can be reused for proving feasibility for each composed event and that is enough to verify this property. From [5]:*

$$\begin{aligned} FIS_{evt1}: \quad & I_1(v_1) \land G_1(p_1, v_1) \vdash \exists v_1' \cdot (S_1(p_1, v_1, v_1')) && (2)\\ FIS_{evt2}: \quad & I_2(v_2) \land G_2(p_2, v_2) \vdash \exists v_2' \cdot (S_2(p_2, v_2, v_2')) && (3)\\ FIS_{evt1\|evt2}: \quad & I_{CM}(v_0, v_1, v_2) \land I_1(v_1) \land I_2(v_2) \land G_1(p_1, v_1) \land G_2(p_2, v_2) && (4)\\ & \vdash \exists v_1', v_2' \cdot (S_1(p_1, v_1, v_1') \land S_2(p_2, v_2, v_2')). \end{aligned}$$

Assume: FIS_{evt1} *and* FIS_{evt2}.
Prove: $FIS_{evt1\|evt2}$.

Proof. Assume the hypotheses of $FIS_{evt1\|evt2}$.

$$\begin{aligned} & I_{CM}(v_0, v_1, v_2) \\ & I_1(v_1) \land G_1(p_1, v_1) && (5)\\ & I_2(v_2) \land G_2(p_2, v_2). && (6) \end{aligned}$$

Prove: $\exists v_1', v_2' \cdot (S_1(p_1, v_1, v_1') \land S_2(p_2, v_2, v_2'))$. The proof proceeds as follows:

$$\begin{aligned} & \exists v_1', v_2' \cdot (S_1(p_1, v_1, v_1') \land S_2(p_2, v_2, v_2')) \\ & \equiv \exists v_1' \cdot (S_1(p_1, v_1, v_1')) \land \exists v_2' \cdot (S_2(p_2, v_2, v_2')) && \{\text{disjoint v1 and v2}\}\\ & \Leftarrow (FIS_{evt1} \land FIS_{evt2}). && \{(2)+(5),(3)+(6)\} \end{aligned}$$

Another consistency PO is invariant preservation. In the composed machine, invariant preservation PO INV_{CM} corresponds to the invariant preservation in all events from the individual machines that are composed. The invariant preservation proof obligation for the composed event $evt1 \parallel evt2$ is $INV_{evt1\|evt2}$.

Theorem 3. *For each invariant i from the set of invariants I in a composed machine, composition invariant* $I_{CM}(v_0, v_1, v_2)$ *needs to be verified. From [5]:*

$$\begin{aligned} INV_{evt1}: \quad & I_1(v_1) \land G_1(p_1, v_1) \land S_1(p_1, v_1, v_1') \vdash i_1(v_1') && (7)\\ INV_{evt2}: \quad & I_2(v_2) \land G_2(p_2, v_2) \land S_2(p_2, v_2, v_2') \vdash i_2(v_2') && (8)\\ INV_{evt1\|evt2}: \quad & I_{CM}(v_0, v_1, v_2) \land I_1(v_1) \land I_2(v_2) \\ & \land G_1(p_1, v_1) \land G_2(p_2, v_2) \\ & \land S_1(p_1, v_1, v_1') \land S_2(p_2, v_2, v_2') \\ & \vdash i_1(v_1') \land i_2(v_2') \land i_{CM}(v_0, v_1', v_2') \end{aligned}$$

Assume: INV_{evt1} *and* INV_{evt2}.
Prove: $INV_{evt1\|evt2}$.

Proof. Assume the hypotheses of $INV_{evt1\|evt2}$.

$$\begin{aligned} & I_{CM}(v_0, v_1, v_2) \\ & I_1(v_1) \land G_1(p_1, v_1) \land S_1(p_1, v_1, v_1') && (9)\\ & I_2(v_2) \land G_2(p_2, v_2) \land S_2(p_2, v_2, v_2') && (10) \end{aligned}$$

Prove: $i_1(v_1') \land i_2(v_2') \land i_{CM}(v_0, v_1', v_2')$. The proof proceeds as follows:

$$\begin{aligned} & i_1(v_1') \land i_2(v_2') \land i_{CM}(v_0, v_1', v_2') \\ & \Leftarrow INV_{evt1} \land INV_{evt2} \land i_{CM}(v_0, v_1', v_2'). && \{(7)+(9),(8)+(10)\} \end{aligned}$$

Well-definedness for expressions (guards, actions, invariants, etc) needs to be verified. These are verified by means of POs in Event-B [16]. For composed machines, well-definedness POs are only generated for $I_{CM}(v_0, v_1, v_2)$. Other expressions are verified in the individual machines.

Refinement Refinement POs are required when the composed machine refines an abstract machine. Machine M_0 with variables v_0, invariant $I_0(v_0)$ and abstract event evt_0 is refined by composed machine CM defined by machines M_1 with variables w_1, invariant $I_1(w_1)$, event evt_1 and M_2 (w_2 ; $I_2(w_2)$; evt_2) and composition invariant $J_{CM}(v_0, w_1, w_2)$. The composed event $evt1 \parallel evt2$ refines the abstract event evt_0. A general refinement PO (REF$_{evti}$) for a machine M refining event $evti$ follows from:

$$\begin{aligned} REF_{evti} \mathrel{\widehat{=}} & I_i(v_i) \wedge J_i(v_i, w_i) \wedge H_i(q_i, w_i) \wedge T_i(q_i, w_i, w_i') \\ & \vdash \exists v_i' \cdot G_i(v_i) \wedge S_i(p_i, v_i, v_i') \wedge J_i(v_i', w_i') \end{aligned} \tag{11}$$

Theorem 4. *For each composed event $evt1 \parallel evt2$, refining abstract event $evt0$ through (gluing) composition invariant in a composed machine, the refinement REF PO consists in proving the guard strengthening of abstract guards, proving the simulation of the abstract variables (v_0') and preserving the gluing invariant ($J_{CM}(v_0', w_1', w_2')$). From [5] and* (11)*:*

$$INV_{evt1} : \quad I_1(w_1) \wedge H_1(q_1, w_1) \wedge T_1(q_1, w_1, w_1') \vdash i_1(w_1') \tag{12}$$

$$INV_{evt2} : \quad I_2(w_2) \wedge H_2(q_2, w_2) \wedge T_2(q_2, w_2, w_2') \vdash i_2(w_2') \tag{13}$$

$$\begin{aligned} REF_{evt0 \sqsubseteq (evt1 \parallel evt2)} : \quad & I_0(v_0) \wedge I_1(w_1) \wedge I_2(w_2) \wedge J_{CM}(v_0, w_1, w_2) \\ & \wedge H_1(q_1, w_1) \wedge H_2(q_2, w_2) \wedge T_1(q_1, w_1, w_1') \wedge T_2(q_2, w_2, w_2') \\ & \vdash \exists v_0' \cdot G_0(p_0, v_0) \wedge S_0(p_0, v_0, v_0') \\ & \wedge I_1(w_1') \wedge I_2(w_2') \wedge J_{CM}(v_0', w_1', w_2'). \end{aligned}$$

Assume: INV_{evt1} (12) *and INV_{evt2}* (13)*.*
Prove: $REF_{evt0 \sqsubseteq (evt1 \parallel evt2)}$.

Proof. Assume the hypotheses of $REF_{evt0 \sqsubseteq (evt1 \parallel evt2)}$. Prove: $\exists v_0' \cdot G_0(p_0, v_0) \wedge S_0(p_0, v_0, v_0') \wedge I_1(w_1') \wedge I_2(w_2') \wedge J_{CM}(v_0', w_1', w_2')$. The proof proceeds as follows:

$$\begin{aligned} & \exists v_0' \cdot G_0(p_0, v_0) \wedge S_0(p_0, v_0, v_0') \\ & \quad \wedge I_1(w_1') \wedge I_2(w_2') \wedge J_{CM}(v_0', w_1', w_2') \\ \equiv\ & G_0(p_0, v_0) \wedge I_1(w_1') \wedge I_2(w_2') \\ & \quad \wedge \exists v_0' \cdot (S_0(p_0, v_0, v_0') \wedge J_{CM}(v_0', w_1', w_2')) \qquad \{\wedge \text{ goal; } v_0, w_1', w_2' \text{ are free variables}\} \\ \equiv\ & G_0(p_0, v_0) \\ & \quad \wedge \exists v_0' \cdot (S_0(p_0, v_0, v_0') \wedge J_{CM}(v_0', w_1', w_2')) \qquad \{\text{from (12) and (13)}\} \end{aligned}$$

These are the required POs to verify composed machines. Next we show that composed machines are monotonic which allows to further refine individual machines preserving composition.

4.3 Monotonicity of Shared Event Composition for Composed Machines

An important property of the shared event composition in Event-B is *monotonicity*. We prove it by means of refinement POs confirming the result described by Butler [13] using actions systems and CSP. Figure 5 shows abstract component specification $M1$ composed with other component specification $N1$, creating a composed model $M1 \parallel N1$. $M1$ is refined by $M2$ and $N1$ by $N2$ respectively. Once we compose specifications $M1$ and $N1$, discharge the required composed POs, $M1$ and $N1$ can be refined individually while the composition properties are preserved without the need to recompose refinements $M2$ and $N2$. We want

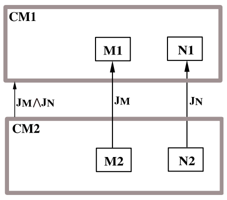


Fig. 5. Refinement of composed machine $CM1 \mathrel{\widehat{=}} M1 \parallel N1$ by $CM2 \mathrel{\widehat{=}} M2 \parallel N2$

to formally prove the monotonicity property through refinement POs between composed machines. Therefore if the refinement POs hold between $CM1$ and $CM2$ then $CM1$: $CM1 \sqsubseteq CM2$. Events evt_{M1} in machine $M1$ and evt_{M2} in machine $M2$ are represented as:

$$evt_{M1} \mathrel{\widehat{=}} \textbf{ANY } p_M \textbf{ WHERE } G_M(p_M, v_M) \textbf{THEN } S_M(p_M, v_M, v'_M) \textbf{ END} \tag{14}$$

$$evt_{M2} \mathrel{\widehat{=}} \textbf{ANY } q_M \textbf{ WHERE } H_M(q_M, w_M) \textbf{THEN } T_M(q_M, w_M, w'_M) \textbf{ END} \tag{15}$$

The gluing invariant of the refinement between M1 and M2 is expressed as $J_M(v_M, w_M)$ relating the states of M1 and M2: $M1 \sqsubseteq_{J_M} M2$. We can derive the refinement PO between $M2$ and $M1$ for the concrete event evt_{M2} refining abstract event evt_{M1}.

$$\begin{aligned} REF_{evt_{M1} \sqsubseteq evt_{M2}}: \quad & I_M(v_M) \land J_M(v_M, w_M) \land G_M(p_M, v_M) \land H_M(q_M, w_M) \\ & \land S_M(p_M, v_M, v'_M) \land T_M(q_M, w_M, w'_M) \\ & \vdash \exists v'_M \cdot G_M(p_M, v_M) \land S_M(p_M, v_M, v'_M) \land J_M(v'_M, w'_M). \end{aligned} \tag{16}$$

The refinement PO between $N2$ and $N1$ is similar. We refine an abstract event in $CM1$ by a concrete one in $CM2$ and verify that the refinement POs for each individual machine hold for the composition. Event evt_{M1} from machine $M1$ and event evt_{N1} from machine $N1$ are composed, resulting in the abstract composed event $evt_{M1} \parallel evt_{N1}$ in $CM1$ from Fig. 5. The gluing invariant relating the states of $CM1$ and $CM2$ is expressed as the conjunction of the gluing invariants between ($M1$ and $M2$) and ($N1$ and $N2$):

$$J_{CM}(v_M, v_N, w_M, w_N) = J_M(v_M, w_M) \land J_N(v_N, w_N) \tag{17}$$

Theorem 5. *The refinement POs for composed machines is expressed as the conjunction of the refinement POs for the individual machines. Therefore the monotonicity property holds if the refinement POs of individual machines hold. The refinement PO between concrete composed event $evt_{M2} \parallel evt_{N2}$ and abstract composed event $evt_{M1} \parallel evt_{N1}$ is expressed as:*

$$\begin{aligned} REF_{(evt_{M1}\parallel evt_{N1})\sqsubseteq(evt_{M2}\parallel evt_{N2})}: \quad & I_M(v_M) \land I_N(v_N) \land J_{CM}(v_M, v_N, w_M, w_N) \\ & \land H_M(q_M, w_M) \land H_N(q_N, w_N) \\ & \land T_M(q_M, w_M, w'_M) \land T_N(q_N, w_N, w'_N) \\ & \vdash \exists v'_M, v'_N \cdot G_M(p_M, v_M) \land G_N(p_N, v_N) \\ & \land S_M(p_M, v_M, v'_M) \land S_N(p_N, v_N, v'_N) \\ & \land J_{CM}(v'_M, v'_N, w'_M, w'_N). \end{aligned} \tag{18}$$

Assume: $REF_{evt_{M1}\sqsubseteq evt_{M2}}$ *and* $REF_{evt_{N1}\sqsubseteq evt_{N2}}$.
Prove: $REF_{(evt_{M1}\parallel evt_{N1})\sqsubseteq(evt_{M2}\parallel evt_{N2})}$.

Proof. Assume the hypotheses of $REF_{(evt_{M1}\parallel evt_{N1})\sqsubseteq(evt_{M2}\parallel evt_{N2})}$.

$$J_{CM}(v_M, v_N, w_M, w_N) \equiv J_M(v_M, w_M) \land J_N(v_N, w_N) \qquad \{\text{expanding } J_{CM} \text{ from (17)}\}$$

$$I_M(v_M) \land H_M(q_M, w_M) \land T_M(q_M, w_M, w'_M) \tag{19}$$

$$I_N(v_N) \land H_N(q_N, w_N) \land T_N(q_N, w_N, w'_N) \tag{20}$$

Prove: $\exists v'_M, v'_N \cdot G_M(p_M, v_M) \land G_N(p_N, v_N) \land S_M(p_M, v_M, v'_M) \land S_N(p_N, v_N, v'_N) \land J_{CM}(v'_M, v'_N, w'_M, w'_N)$. The proof proceeds as follows:

$$\begin{aligned} & \exists v'_M, v'_N \cdot G_M(p_M, v_M) \land G_N(p_N, v_N) \\ & \quad \land S_M(p_M, v_M, v'_M) \land S_N(p_N, v_N, v'_N) \\ & \quad \land J_M(v'_M, w'_M) \land J_N(v'_N, w'_N) && \{\text{expanding } J_{CM} \text{ from (17)}\} \\ & \equiv \exists v'_M \cdot G_M(v_M) \land S_M(p_M, v_M, v'_M) \land J_M(v'_M, w'_M) \\ & \quad \land \exists v'_N \cdot G_N(v_N) \land S_N(p_N, v_N, v'_N) \land J_N(v'_N, w'_N) && \{\text{disjoint } v'_M, v'_N\} \\ & \Leftarrow REF_{evt_{M1}\sqsubseteq evt_{M2}} \land REF_{evt_{N1}\sqsubseteq evt_{N2}} && \{(16)+(19),(16)+(20)\} \end{aligned}$$

We also need to prove the monotonicity for single (non-composed) events that appear at both levels of abstraction. We shall prove it using machines $M1$ and $CM2$. In this case, the gluing invariant described in (17) does not use neither the variables (v_N) neither the invariants(I_N) neither events (evt_{N1}) from $N1$. Therefore it can be simplified and rewritten as:

$$J_{CM}(v_M, w_M, w_N) = J_M(v_M, w_M) \land J_N(w_N) \tag{21}$$

Deriving from (21), the goal of $INV_{evt_{M2}\parallel evt_{N2}}$ can be expanded to:

$$j_{CM}(v'_M, w'_M, w'_N) \equiv j_M(v'_M, w'_M) \land j_N(w'_N) \tag{22}$$

where j_M and j_N correspond to each invariant from the set of gluing invariants J_M and J_N respectively.

Theorem 6. *An individual event evt_{M1} in machine $M1$ is refined by a composed event $evt_{M2} \parallel evt_{N2}$ in composed machine $CM2$. The monotonicity is preserved* *if the refinement PO between $M1$ and $M2$ hold in conjunction with the gluing invariant preservation PO for the composed event $evt_{M2} \parallel evt_{N2}$. The refinement PO between concrete composed event $evt_{M2} \parallel evt_{N2}$ and abstract non-composed event evt_{M1}:*

$$\begin{aligned} REF_{evt_{M1} \sqsubseteq (evt_{M2} \parallel evt_{N2})} : \quad & I_M(v_M) \land J_{CM}(v_M, w_M, w_N) \\ & \land H_M(q_M, w_M) \land H_N(q_N, w_N) \\ & \land T_M(q_M, w_M, w'_M) \land T_N(q_N, w_N, w'_N) \\ & \vdash \exists v'_M \cdot G_M(p_M, v_M) \land S_M(p_M, v_M, v'_M) \\ & \land J_{CM}(v'_M, w'_M, w'_N). \end{aligned} \tag{23}$$

Assume: $REF_{evt_{M1} \sqsubseteq evt_{M2}}$ and $INV_{evt_{M2} \parallel evt_{N2}}$.
Prove: $REF_{evt_{M1} \sqsubseteq (evt_{M2} \parallel evt_{N2})}$.

Proof. Assume the hypotheses of $REF_{evt_{M1} \sqsubseteq (evt_{M2} \parallel evt_{N2})}$.

$$\begin{aligned} & J_{CM}(v_M, w_M, w_N) \equiv J_M(v_M, w_M) \land J_N(w_N) \quad \{\text{expanding } J_{CM} \text{ from (21)}\}. \\ & I_M(v_M) \land H_M(q_M, w_M) \land T_M(q_M, w_M, w'_M) \quad (24) \\ & H_N(q_N, w_N) \land T_N(q_N, w_N, w'_N) \end{aligned}$$

And the hypotheses of $INV_{evt_{M2} \parallel evt_{N2}}$:

$$\begin{aligned} & J_{CM}(v_M, w_M, w_N) \equiv J_M(v_M, w_M) \land J_N(w_N) \quad \{\text{expanding } J_{CM} \text{ from (21)}\} \\ & I_M(v_M) \land H_M(q_M, w_M) \land T_M(q_M, w_M, w'_M) \\ & W_2(v'_M, w_M, w_N, q_M, q_N, w'_M, w'_N) \quad (25) \\ & H_N(q_N, w_N) \land T_N(q_N, w_N, w'_N) \quad (26) \end{aligned}$$

Prove: $\exists v'_M \cdot G_M(p_M, v_M) \land S_M(p_M, v_M, v'_M) \land J_{CM}(v'_M, w'_M, w'_N)$. The proof proceeds as follows:

$$\begin{aligned} & \exists v'_M \cdot G_M(p_M, v_M) \land S_M(p_M, v_M, v'_M) \\ & \quad \land J_{CM}(v'_M, v'_N, w'_M, w'_N) \\ \equiv \; & \exists v'_M \cdot G_M(p_M, v_M) \land S_M(p_M, v_M, v'_M) \\ & \quad \land J_M(v'_M, w'_M) \land J_N(w'_N) && \{\text{expanding } J_{CM} \text{ from (21)}\} \\ \equiv \; & \exists v'_M \cdot G_M(p_M, v_M) \land S_M(p_M, v_M, v'_M) \land J_M(v'_M, w'_M) \\ & \quad \land J_N(w'_N) && \{\text{disjoint } v'_M\} \\ \Leftarrow \; & REF_{evt_{M1} \sqsubseteq evt_{M2}} \\ & \quad \land J_N(w'_N) && \{(16)+(24)\} \\ \Leftarrow \; & REF_{evt_{M1} \sqsubseteq evt_{M2}} \\ & \quad \land INV_{evt_{M2} \parallel evt_{N2}} && \{(22)+(25)+(26)\} \end{aligned}$$

New events can be added during refinement. They must respect the refinement POs. The refinement PO proof for new events is similar to the previous cases but applied to a single event refined by composed event. Due to the lack of space we do not present it here.

5 Decomposition Guideline

Based on the work developed for composition, its properties and the inverse relation between composition and decomposition, we develop a methodology to partition models in a shared event style. As described in Sect. 3.2, in a shared event decomposition approach, the variables of a system are separated into different sub-components and consequently the rest of the system is decomposed. We present the steps that are required in order to process during decomposition.

Variables: From the modeller's point of view, the decomposition starts by defining which sub-components are generated. The following step is to define the partition of variables over the sub-components. The rest of the model decomposition (events, parameters, invariants, contexts) is a consequence of the variables allocation as defined below.

Invariants: The decomposition of the invariants depends on the scope of the variables. It is still not very clear which invariants should be retained in the decomposition except the ones related with variable type definition. Further invariants need more study to determine their partition. It seems that they should depend on the input of the user since they might be a constraint of the composed component and not a requirement of the sub-component. When an invariant clause is required but uses variables placed outside the scope of a sub-component, a further refinement of the composed component might be required to make an explicit separation of the variables. An option is to duplicate variables, suggested by Butler [17,18].

Events: The partition of the events depends on the partition of the variables. When the decomposition occurs, parameters are shared between the decomposed events. But the guards referring to that parameter can be different in each decomposed event. The guard of a decomposed event inherits the guard on the composed event according to the variable partition. For example, let us consider event $e1$:

$$e1 \mathrel{\widehat{=}} \textbf{WHEN } c = \text{TRUE } \textbf{THEN } a := b \;\; \| \; c := \text{FALSE}$$

where variables a and b are of type $DATA$ and variable c is a Boolean. This event is enabled when c is TRUE and results in a being assigned the value of b and this event being disabled by assigning c to FALSE. If this event is decomposed such that variable a belongs to one sub-component and variables b and c belong to another, then the action $b := m$ needs to be split. Although the original event does not have parameters, the decomposed events have a new parameter p. During the decomposition, that assignment is divided into three steps and a parameter p is introduced:

$$a := b \Leftrightarrow p \in DATA \wedge p = b \wedge a := p$$

Parameter p receives the value of variable b. Then the value of p is assigned to variable a. The resulting decomposed events are:

$$e1' \mathrel{\widehat{=}} \textbf{ANY } p \textbf{ WHERE } p = b \wedge c = \text{TRUE } \textbf{THEN } c := \text{FALSE}$$

$$e1'' \mathrel{\widehat{=}} \textbf{ANY } p \textbf{ WHERE } p \in DATA \textbf{ THEN } a := p$$

These corresponds to the value passing of parallel events similar to suggested by Butler [13] for action systems based on CSP: for event *e1'*, parameter p has a output behaviour as it is written by the value of b; in event $e1''$, parameter p has an input behaviour has the value is read and assigned to variable a.

The events in the sub-components resulting from the decomposition maintain the interface of the original events, preserving the parts corresponding to the variables that belongs to each sub-component.

6 File Access Management case study

A distributed system is presented where a system is decomposed into two smaller parts. A specification of a file management system is developed: files containing *DATA* can be created, read, overwritten, deleted and sent to other users. Each file has an owner. The owners are users with clearance level ranging from 1 to 10 where 10 is the highest level. A *super* user exists with clearance level 10. Moreover, files have a classification level varying from 1 to 10. Permission is needed in order to read, modify or delete a file. When the permission is granted, the requested action can take place.

Machine $FileAccessManagement$ contains variables *user*, *file*, *fileData* (contains the data of each file) and *fileStatus* (defines the status of a file operation and can have the states *SUCCESS* or *FAILED*). When a file is created or sent, variable *fileStatus* is updated accordingly to the result of the operation. The status of a file must be reset (in event *clearFileStatus*) to allow a new operation in the same file. The access management is defined by variables *userClearanceLevel*, *permission*, *fileClassification* and *fileOwner*. A user can change the clearance of another user as long as the former has a clearance level superior to the latter as described in event $modifyUser$ (guard *grd3* in Fig. 6(b)). For all the other operations, permission is required and it is granted by the non-deterministic action in event *requestPermission*. When a permission is granted, a file can be read, modified, deleted or sent to another user. A file can only be modified by users with a clearance level superior to the file classification (guard *grd8* in event *overwriteFile*). To delete a file, described in event *deleteFile*, the user must be the owner of the file or be the *super user* as described by guard *grd5*.

Our intention is to separate the management of permissions (administrative task) from the modification of the files in the disk (writing, reading tasks). The result are two sub-components, *AccessMng* and *FileMng* that deal with different parts of the system. An advantage of this decomposition is that it becomes easier to define specific properties to each part without additional constraints of the other part. For instance, an administrative task of *AccessManagement* is to have a quota of disk per user which is irrelevant to *FileMng*. Overwriting a file in the disk is relevant to *FileMng* but not to *AccessMng* that deals

```
variables userClearanceLevel permission
          fileClassification fileOwner user file
          fileData fileStatus

invariants
  @inv1 file ⊆ FILE
  @inv2 user ⊆ USER
  @inv3 userClearanceLevel ∈ user → ClearanceLevel
  @inv4 permission ∈ PERMISSION
  @inv5 fileClassification ∈ file → Classification
  @inv6 fileOwner ∈ file → user
  @inv7 fileData ∈ file → DATA
  @inv8 fileStatus ∈ file ⇸ STATUS
  @inv9 ran(fileStatus) ⊆ {SUCCESS, FAILED}
  @inv10 fileOwner ∈ file → user
  @inv11 ∀f·f ∈ file ⇒
         userClearanceLevel(fileOwner(f)) > fileClassification(f)
```

(a)

```
event addUser
  any uu // changed user
      masterUser // user who will make the change to uu
      newUserClearanceLevel // new user ClearanceLevel
  where
    @grd1 uu ∈ dom(userClearanceLevel)
    @grd2 newUserClearanceLevel ∈ ClearanceLevel
    @grd3 newUserClearanceLevel < userClearanceLevel(uu)
    @grd4 masterUser ≠ uu
    @grd5 uu ≠ super
    @grd6 ∀f·f ∈ dom(fileClassification) ∧ fileOwner(f)=uu
          ⇒ newUserClearanceLevel>fileClassification(f)
    @grd7 uu ∉ user
    @grd8 masterUser ∈ user
  then
    @act1 userClearanceLevel(uu)≔ newUserClearanceLevel
    @act2 user ≔ user ∪ {uu}
end

event modifyUser
  any uu // changed user
      masterUser // user who will make the change to uu
      newUserClearanceLevel // new user ClearanceLevel
  where
    @grd1 uu ∈ dom(userClearanceLevel)
    @grd2 newUserClearanceLevel ∈ ClearanceLevel
    @grd3 newUserClearanceLevel < userClearanceLevel(uu)
    @grd4 masterUser ≠ uu
    @grd5 uu ≠ super
    @grd6 ∀f·f ∈ dom(fileClassification) ∧ fileOwner(f)=uu
          ⇒ newUserClearanceLevel>fileClassification(f)
  then
    @act1 userClearanceLevel(uu)≔ newUserClearanceLevel
end

vent overwriteFile
  any ff dd cl u
  where
    @grd1 ff ∈ file
    @grd2 dd ∈ DATA
    @grd3 dd ≠ fileData(ff)
    @grd4 u∈dom(userClearanceLevel)
    @grd5 cl ∈ Classification
    @grd6 permission = ALLOWED
    @grd7 ff ∈ dom(fileClassification)
           ⇒ cl = fileClassification(ff)
    @grd8 userClearanceLevel(u)>cl
  then
    @act1 fileData(ff)≔dd
    @act2 fileClassification(ff)≔ cl
    @act3 permission ≔ OFF
    @act4 fileOwner(ff)≔ u
end

event deleteFile
  any ff // file to be deleted
      u // user executes the action
  where
    @grd1 ff ∈ file
    @grd2 u ∈ user
    @grd3 permission = ALLOWED
    @grd4 ff ∈ dom(fileOwner)
    @grd5 u ∈ {super,fileOwner(ff)}
  then
    @act1 file≔file\{ff}
    @act2 fileData≔{ff}⩤fileData
    @act3 fileStatus≔{ff}⩤fileStatus
    @act4 fileClassification≔{ff}⩤fileClassification
    @act5 permission ≔ OFF
    @act6 fileOwner≔{ff}⩤fileOwner
end

event sendFile
  any ff recipient u fs cl
  where
    @grd1 ff ∈ file
    @grd2 u ∈ user
    @grd3 recipient ∈ user
    @grd4 ff ∉ dom(fileStatus)
    @grd5 fs ∈ {SUCCESS,FAILED}
    @grd6 u ≠ recipient
    @grd7 u∈dom(userClearanceLevel)
    @grd8 cl ∈ Classification
    @grd9 permission = ALLOWED
    @grd10 ff ∈ dom(fileClassification)
            ⇒ cl = fileClassification(ff)
    @grd11 userClearanceLevel(u)>cl
  then
    @act1 fileStatus(ff) ≔ fs
    @act2 fileClassification(ff)≔ cl
    @act3 permission ≔ OFF
    @act4 fileOwner(ff)≔ u
end

event requestPermission
  where
    @grd1 permission ≠ ALLOWED
  then
    @act1 permission:∈ PERMISSION\{OFF}
end

event clearFileStatus
  any ff
  where
    @grd1 ff ∈ dom(fileStatus)
    @grd2 fileStatus(ff)∈{SUCCESS,FAILED}
  then
    @act1 fileStatus ≔ {ff}⩤fileStatus
end
```

(b)

Fig. 6. *FileAccessManagement*: variables, invariants (a) and some events (b)

with the users that are allowed to execute this action is not. Therefore we decompose $FileAccessManagement$ into two sub-components as described in the next section.

6.1 Decomposition *FileAccessManagement*: *AccessMng* and *FileMng*

Following the steps suggested in Sect. 5, the variables of $FileAccessManagement$ need to be allocated to sub-components *AccessMng* and *FileMng* as described in the following table:

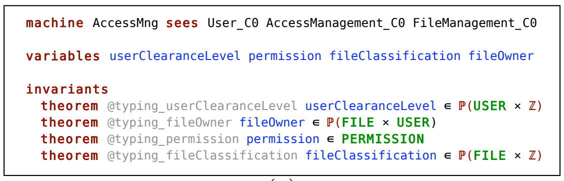
	FileMng	AccessMng
Variables	file,user, fileData,fileStatus	userClearanceLevel,permission, fileOwner,fileClassification

The distribution of events can be seen on the composed machine described in Fig. 7.

```
COMPOSED MACHINE FileAccessManagement
INCLUDES
    AccessMng, FileMng
EVENTS
    addUser
        Combines Events AccessMng.addUser || FileMng.addUser
    modifyUser
        Combines Events AccessMng.modifyUser
    createFile
        Combines Events AccessMng.createFile || FileMng.createFile
    readFile
        Combines Events AccessMng.readFile || FileMng.readFile
    overwriteFile
        Combines Events AccessMng.overwriteFile || FileMng.overwriteFile
    deleteFile
        Combines Events AccessMng.deleteFile || FileMng.deleteFile
    sendFile
        Combines Events AccessMng.sendFile || FileMng.sendFile
    requestPermission
        Combines Events AccessMng.requestPermission
    clearFileStatus
        Combines Events FileMng.clearFileStatus
```

Fig. 7. Composed machine *FileAccessManagement*

Some events are specific to a sub-component: events $modifyUser$ and *requestPermission* belong to *AccessMng* while *clearFileStatus* belongs to *FileMng*. The majority of the events are decomposed between the two sub-components and when they are synchronised and occur in parallel, they refine the original event before the decomposition. The resulting sub-components can be seen in Figs. 8 and 9. Compositon and decomposition are combined when modelling this system: the decomposition partition the model in sub-components based on the variables and the composition expresses how the decomposed events interact. *Silva et al* [19] present a decomposition tool that permits the semi-automatic decomposition in a shared event or shared variable style. Our case

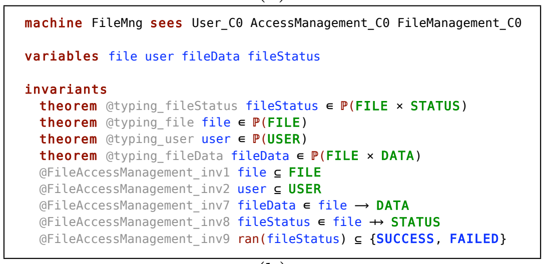


```
machine AccessMng sees User_C0 AccessManagement_C0 FileManagement_C0

variables userClearanceLevel permission fileClassification fileOwner

invariants
  theorem @typing_userClearanceLevel userClearanceLevel ∈ ℙ(USER × ℤ)
  theorem @typing_fileOwner fileOwner ∈ ℙ(FILE × USER)
  theorem @typing_permission permission ∈ PERMISSION
  theorem @typing_fileClassification fileClassification ∈ ℙ(FILE × ℤ)
```

(a)



```
machine FileMng sees User_C0 AccessManagement_C0 FileManagement_C0

variables file user fileData fileStatus

invariants
  theorem @typing_fileStatus fileStatus ∈ ℙ(FILE × STATUS)
  theorem @typing_file file ∈ ℙ(FILE)
  theorem @typing_user user ∈ ℙ(USER)
  theorem @typing_fileData fileData ∈ ℙ(FILE × DATA)
  @FileAccessManagement_inv1 file ⊆ FILE
  @FileAccessManagement_inv2 user ⊆ USER
  @FileAccessManagement_inv7 fileData ∈ file → DATA
  @FileAccessManagement_inv8 fileStatus ∈ file ⇸ STATUS
  @FileAccessManagement_inv9 ran(fileStatus) ⊆ {SUCCESS, FAILED}
```

(b)

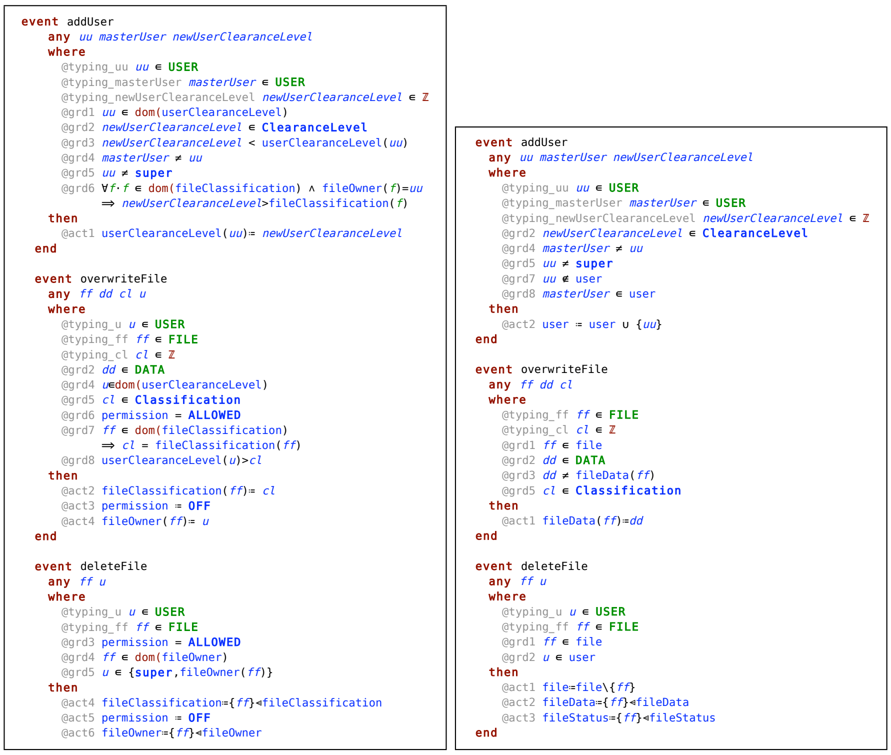
Fig. 8. $AccessMng$ (a) and $FileMng$ (b): variables and invariants

study was run using this tool and we show that we can integrate the shared event decomposition in cooperation with composition (and respective composition tool [20]).

One of the properties of the shared event composition is monotonicity. Therefore sub-components can be further refined independently preserving the verified properties while composed. For instance, machine *AccessMng* can be refined by defining a more deterministic event *requestPermission* based on the kind of operation and the user that intends to execute the operation. For machine *FileMng*, the event *sendFile* can be further refined by introducing a queue where events would be stored before being processed (create a new file own by the recipient of the file). The independent refinement of the sub-components results in a separation of behaviours and properties that can be verified without the interference of other sub-components.

7 Conclusions

Composition allows the interaction of sub-components. Back [21], Abadi and Lamport[22] studied the interaction of components through shared variable composition. Jones [23] also proposes a shared variable composition for VDM by restricting the behaviour of the environment and the operation itself in order to consider the composition valid using rely-guarantee conditions. In Z, composition can be achieved by combining schemas [24] where variables within the same scope cannot have identical names or by views [1] allowing the development of partial specifications that can interact through invariants that relate their state or by operations' synchronisation. Although systems are developed in single machines in classical B, Bellergarde et at [25] suggest a composition by rearranging



```
event addUser
    any uu masterUser newUserClearanceLevel
    where
      @typing_uu uu ∈ USER
      @typing_masterUser masterUser ∈ USER
      @typing_newUserClearanceLevel newUserClearanceLevel ∈ ℤ
      @grd1 uu ∈ dom(userClearanceLevel)
      @grd2 newUserClearanceLevel ∈ ClearanceLevel
      @grd3 newUserClearanceLevel < userClearanceLevel(uu)
      @grd4 masterUser ≠ uu
      @grd5 uu ≠ super
      @grd6 ∀f·f ∈ dom(fileClassification) ∧ fileOwner(f)=uu
            ⇒ newUserClearanceLevel>fileClassification(f)
    then
      @act1 userClearanceLevel(uu)≔ newUserClearanceLevel
  end

  event overwriteFile
    any ff dd cl u
    where
      @typing_u u ∈ USER
      @typing_ff ff ∈ FILE
      @typing_cl cl ∈ ℤ
      @grd2 dd ∈ DATA
      @grd4 u∈dom(userClearanceLevel)
      @grd5 cl ∈ Classification
      @grd6 permission = ALLOWED
      @grd7 ff ∈ dom(fileClassification)
            ⇒ cl = fileClassification(ff)
      @grd8 userClearanceLevel(u)>cl
    then
      @act2 fileClassification(ff)≔ cl
      @act3 permission ≔ OFF
      @act4 fileOwner(ff)≔ u
  end

  event deleteFile
    any ff u
    where
      @typing_u u ∈ USER
      @typing_ff ff ∈ FILE
      @grd3 permission = ALLOWED
      @grd4 ff ∈ dom(fileOwner)
      @grd5 u ∈ {super,fileOwner(ff)}
    then
      @act4 fileClassification≔{ff}⩤fileClassification
      @act5 permission ≔ OFF
      @act6 fileOwner≔{ff}⩤fileOwner
```

```
event addUser
  any uu masterUser newUserClearanceLevel
  where
    @typing_uu uu ∈ USER
    @typing_masterUser masterUser ∈ USER
    @typing_newUserClearanceLevel newUserClearanceLevel ∈ ℤ
    @grd2 newUserClearanceLevel ∈ ClearanceLevel
    @grd4 masterUser ≠ uu
    @grd5 uu ≠ super
    @grd7 uu ∉ user
    @grd8 masterUser ∈ user
  then
    @act2 user ≔ user ∪ {uu}
end

event overwriteFile
  any ff dd cl
  where
    @typing_ff ff ∈ FILE
    @typing_cl cl ∈ ℤ
    @grd1 ff ∈ file
    @grd2 dd ∈ DATA
    @grd3 dd ≠ fileData(ff)
    @grd5 cl ∈ Classification
  then
    @act1 fileData(ff)≔dd
end

event deleteFile
  any ff u
  where
    @typing_u u ∈ USER
    @typing_ff ff ∈ FILE
    @grd1 ff ∈ file
    @grd2 u ∈ user
  then
    @act1 file≔file\{ff}
    @act2 fileData≔{ff}⩤fileData
    @act3 fileStatus≔{ff}⩤fileStatus
end
```

Fig. 9. *AccessMng* (a) and *FileMng* (b): decomposed events *addUser*, *overwriteFile* and *deleteFile*

separated machines and synchronising their operations under feasibility conditions. The behaviour of a component composition is seen as a labelled transition system using weakest preconditions, where a set of authorised transitions are defined. The objective is to verify the refinement of synchronised parallel composition between components but it is limited to finite state transitions and a finite number of components. This work differs from ours as it uses a labelled transition system including a notion of refinement and variable sharing while we use synchronisation and communication in the CSP style. Butler and Walden [26] discuss a combination of action systems and classical B by composing machines using parallel systems in an action system style and preserving the invariants of the individual machines. This approach allows the classical B to derive parallel and distributed systems and since the parallel composition of action system is monotonic, the sub-systems in a parallel composition may be refined independently. This work is closely related to our work with similar underlying semantics and notion of refinement based on CSP. Abrial et al [6] propose a state-based decomposition for Event-B introducing the notion of shared variables and external events. Although it allows variable sharing, this approach is also monotonic but its respective nature is more suitable for parallel programs [27]. Sorge et al [28] propose a feature composition in Event-B and define composition POs to ensure its consistency. In the feature composition approach, exploration of specifications' composition with possible variable sharing (similar to the shared variable style) is allowed but no refinement is defined which differs from our work. Nevertheless similar to our work, sub-components POs are reused to avoid re-proving composition POs.

Our Event-B composition and decomposition is based on the close relation between action systems and Event-B plus the correspondence between action systems and CSP [13]. Shared event composition is proved to be monotonic by means of POs. Refinement in a "top-down" style for developing specifications is allowed. Sub-components interact through event parameters by value-passing and can be further refined. We extend Event-B to support shared event composition, allowing combination and reuse of existing sub-components through the introduction of *composed machines*. Such an approach seems suitable for modelling (distributed) systems. We combine composition and decomposition and suggest a methodology for modelling systems including the verification of properties through the generation of POs and refinement. We do not address the step corresponding to the translation of this composition to an implementation. This study needs to be carried out in the future. A file access management system is decomposed into two independent parts with a separation of their logics: file management and access management. Possible refinement for each sub-component are suggested to carry on this development. Other case studies have been applying composition with success in particular for distributed systems such as the decomposition of a safe metro system.

References

1. Jackson, D.: Structuring Z specifications with views. ACM Trans. Softw. Eng. Methodol. **4**(4) (1995) 365–389
2. Zave, P., Jackson, M.: Conjunction as Composition. ACM Trans. Softw. Eng. Methodol. **2**(4) (1993) 379–411
3. Jones, C.B.: Wanted: a compositional approach to concurrency. In: Programming methodology. Springer-Verlag New York, Inc., New York, NY, USA (2003) 5–15
4. Poppleton, M.: The Composition of Event-B Models. In: ABZ2008: Int. Conference on ASM, B and Z. Volume 5238., Springer LNCS (September 2008) 209–222
5. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
6. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundam. Inf. **77**(1-2) (2007) 1–28
7. Butler, M.: An Approach to the Design of Distributed Systems with B AMN. In: Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212. (1997) 221–241
8. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International Series in Computer Science (1985)
9. Morgan, C.: Of wp and CSP. In: Beauty is our business: a birthday salute to Edsger W. Dijkstra. Springer-Verlag New York, Inc., New York, NY, USA (1990) 319–326

10. Rodin: RODIN project Homepage. `http://rodin.cs.ncl.ac.uk` (September 2008) Online; accessed 27-July-2010.
11. Abrial, J.R., Butler, M.J., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: ICFEM. (2006) 588–605
12. Ralph-Johan R. Back, Kurki-Suonio, R.: Decentralization of Process Nets with Centralized Control. In: PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing, New York, NY, USA, ACM (1983) 131–142
13. Butler, M.J.: A CSP Approach to Action Systems. PhD thesis, Oxford University (1992)
14. Butler, M.: Stepwise Refinement of Communicating Systems. Science of Computer Programming **27**(2) (September 1996) 139–173
15. Butler, M.: Synchronisation-Based Decomposition for Event-B. In: RODIN Deliverable D19 Intermediate report on methodology. (2006) 47–57
16. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. International Journal on Software Tools for Technology Transfer (STTT) (April 2010)
17. Butler, M.: Decomposition Structures for Event-B. Integrated Formal Methods iFM2009 (February 2009) 20–38
18. Butler, M.: Incremental Design of Distributed Systems with Event-B. Marktoberdorf Summer School 2008 Lecture Notes (November 2008)
19. Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition Tool for Event-B. Software: Practice and Experience **41**(2) (February 2011) 199–208
20. Silva, R., Butler, M.: Parallel Composition Using Event-B. `http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B` (July 2009) Online; accessed 27-July-2010.
21. Ralph-Johan R. Back: Refinement Calculus, part II: Parallel and Reactive Programs. In: REX workshop: Proceedings on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness, New York, NY, USA, Springer-Verlag New York, Inc. (1990) 67–93
22. Abadi, M., Lamport, L.: Composing Specifications. In de Bakker, J.W., de Roever, W.P., Rozenberg, G., eds.: Stepwise Refinement of Distributed Systems - Models, Formalisms, Correctness. Volume 430., Berlin, Germany, Springer-Verlag (1989) 1–41
23. Woodcock, J., Dickinson, B.: Using VDM with Rely and Guarantee-Conditions. In: Proceedings of the 2nd VDM-Europe Symposium on VDM—The Way Ahead, New York, NY, USA, Springer-Verlag New York, Inc. (1988) 434–458
24. Spivey, J.M.: The Z Notation: a Reference Manual. Prentice-Hall, Inc. (1989)
25. Bellegarde, F., Julliand, J., Kouchnarenko, O.: Synchronized Parallel Composition of Event Systems in B. In: ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B, London, UK, Springer-Verlag (2002) 436–457
26. Butler, M., Waldén, M.: Distributed System Development in B. Technical Report TUCS-TR-53, Turku Centre for Computer Science (14, 1996)
27. Hoang, T., Abrial, J.R.: Event-B Decomposition for Parallel Programs. Abstract State Machines, Alloy, B and Z (2010) 319–333
28. Sorge, J., Poppleton, M., Butler, M.: A Basis for Feature-Oriented Modelling in Event-B. In: ABZ2010. (February 2010)