Compositional Symmetric Sharing in B

Martin Büchi and Ralph Back

Åbo Akademi University
Turku Centre for Computer Science
Lemminkäisenkatu 14A, 20520 Turku, Finland
{Martin.Buechi, Ralph.Back}@abo.fi,
http://www.abo.fi/~{mbuechi, backrj}

Abstract. Sharing between B constructs is limited, both on the specification and the implementation level. The limitations stem from the single writer/multiple readers paradigm, restricted visibility of shared variables, and structural constraints to prevent interference. As a consequence, applications with inherent sharing requirements have to either be described as large monolithic constructs or be underspecified, leading to a loss of modularity respectively certain desirable properties being unprovable.
We propose a new compositional symmetric shared access mechanism based on roles describing rely/guarantee conditions. The mechanism provides for multiple writers on shared constructs, visibility of shared variables in the accessors' invariants, and controlled aliasing. Use is uniform in machines, refinements, and implementations. Sharing is compositional: all proof obligations are local and do not require knowledge of the other accessors' specifications, let alone their or the shared construct's implementation.
Soundness of the mechanism is established by flattening.

1 Introduction

The B method provides support for modularization and, herewith, for information hiding, compositionality of module operations, reusability of modules, and decomposition of proofs [4, 5]. Modules can be combined using a number of different mechanisms. Refinement being 'almost' monotonic with respect to the composition mechanisms, most proof obligations arise on a per module base. The few additional restriction on the global structure can be checked automatically. In this compositional approach, we can focus on a part of a large system, establish desired properties for this part, and be guaranteed that these properties hold in the complete system.

To achieve compositionality [8] and independent refinement, sharing is restricted in B. Sharing is based on the single writer/multiple readers paradigm. If several constructs access a shared construct, only one accessor (writer) can modify the state of the shared construct. The other accessors (readers) are limited to read-only access, respectively to calling inquiry operations. To ensure that invariants cannot be invalidated, only the single writer is allowed to reference variables of the shared construct in its invariant. Because of these limitations, applications with inherent sharing requirements cannot be handled satisfactorily, as described in Sect. 2.2.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 431–451, 1999.

We introduce a new sharing mechanism that overcomes the single writer and the variable visibility restrictions. Multiple constructs can have write access to a shared construct and reference shared variables in their invariants. The mechanism is compositional: all proof obligations arise on a per module base and only a few automatically checkable restrictions on the sharing graph are required for global correctness. The key element are freely specifiable accessor roles, which determine how the different accessors can use the shared construct. Adherence to these role specifications guarantees that the accessors do not invalidate each others invariants.

Role specifications can be considered as guarantee conditions in the sense of Cliff Jones [15] with the rely conditions being given by the other roles. Rely/guarantee conditions (also known as assumption/commitment specifications) have been developed as a compositional proof method for shared variable and message-passing concurrency with interleaving semantics. This paper shows that the same theory is also applicable to modular sequential systems with sharing.

Section 2 reviews the existing sharing mechanisms and illustrates a shortcoming on a concrete example. Throughout the paper we use numbered variations of the same example. In Sect. 3 we take the problem to its roots, analyzing the reasons for the existing restrictions. Section 4 introduces the new mechanism. We provide further details of the sharing mechanism in Sect. 5. In Sect. 6 we list the complete syntax, the proof obligations, the visibility rules, and the well-formedness criteria for the composition graph. Using flattening of constructs, we prove the soundness of the proposed mechanism in Sect. 7. Sect. 8 lists related work and draws the conclusions.

2 The Problem

2.1 Review of Existing Composition Mechanisms

B has three different constructs: machines, refinements, and implementations, distinguished by different syntactic restrictions. Machines express original specifications. Refinements are intermediate constructs. An implementation denotes the end of a refinement chain and contains executable code. In addition to behavioral specifications, refinements and implementations contain data refinement relations in form of gluing invariants. Machines can be parameterized. Parameters are instantiated by the single writer.

The B method has four mechanisms to compose constructs. The different mechanisms can be used in different constructs. The target of a composition is always a machine.

INCLUDES The *INCLUDES* clause can appear in machines and refinements. It can be understood as textual inclusion with the restriction that variables of the included machine can only be modified indirectly through operations of the included machine so that the invariant of the included machine is preserved. The including construct instantiates the parameters of the included machine and can reference variables of the included machine in the invariant. The including construct becomes the focus of refinement, the included construct doesn't have to be implemented unless it is also imported.

USES The *USES* clause can only appear in machines. It provides for limited sharing on the specification level. Any number of machines can use a shared machine. All using and the used machine must be included into a common machine, which becomes the focus of refinement. The using machines cannot be refined. They have read-only access to the shared machine and can reference shared variables in their invariants. To guarantee that the including machine, which is the only writer, does not invalidate the invariants of the using machines, the using machines' invariants have to be proved upon inclusion.

SEES The *SEES* clause can appear in any construct. It provides for read only access to a shared machine. Variables of the seen machine cannot be referenced in the invariant of the seeing construct. Without this restriction, the invariant of the seeing machine could be invalidated by the construct with write access to the seen machine.

IMPORTS The *IMPORTS* clause can only appear in implementations. The importing machine instantiates the parameters of the imported machine, can call both inquiry and modification operations and can reference variables of the imported machine in its invariant. Imported machines can be seen by any number of other constructs.

Summary The *INCLUDES* and *USES* clauses can be considered as weak or syntactic relations [5]. Their aim is to combine text of machine specifications; this structure is not reflected in subsequent refinements or in the final implementation. *SEES* and *IMPORTS* on the other hand are strong relations as the shared code will remain visible as a module in the final implementation.

2.2 A Problem with the Existing Mechanisms

In this subsection, we illustrate a shortcoming of the existing sharing mechanisms with a concrete example. The example's main characteristic are its inherent sharing requirements.

We consider a control system for a manufacturing plant consisting of various devices, such as robot arms and conveyor belts. Each device is controlled by its own software module, the central *Run* operation of which is periodically called by a scheduler. Whenever a device controller notices an error, the device is stopped and an alarm is registered in a central database. The plant operator can list the active alarms on the screen and deactivate alarms after fixing their cause. The devices check whether all their alarms have been deactivated and if so resume work.

The database is shared between all the device controllers and the monitoring console. They all need both read and write access: the device controllers need to check for active alarms and enter new alarms, the monitoring console needs to list active alarms and change the activity status of alarms.

How do we specify, refine, and implement such an architecture using B's existing composition mechanisms? Let us first look at the specification. Since the devices are to a certain degree independent and since multiple instances of the same device type can

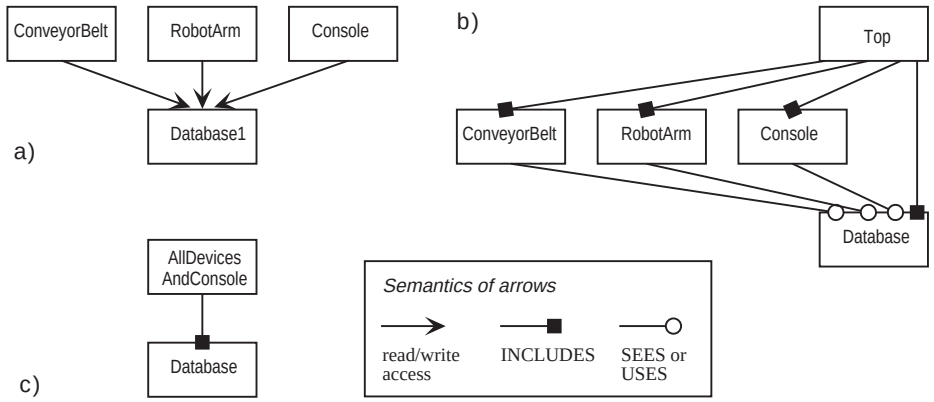

Fig. 1. Specification Approaches

exist, it makes sense to specify them modularly using separate machines. Likewise, the database is captured by a separate machine, which is accessed by the device controllers and the monitoring console (Fig. 1 a). As remarked above, all accessors need both read and write access to the shared database. However, the existing sharing mechanisms are limited by the single writer paradigm (Sect. 2.1).

This sharing architecture is possible with *SEES* or *USES*, if the called operations, such as *NewAlarm* in *Database1*, are specified as inquiry only – although their implementations modify the concrete state:

MACHINE Database1
OPERATIONS
NewAlarm(type) $\widehat{=}$ **PRE** type $\in$ NAT **THEN skip END**;
bb $\leftarrow$ ActiveAlarms $\widehat{=}$ **BEGIN** bb :$\in$ BOOL **END**;
...
END

Unfortunately, this underspecification precludes any sensible reasoning. Becauser there is no set of active alarms in *Database1*, we cannot express the fact that the conveyor belt is only running if none of its alarms are active. Likewise, we cannot prove that an alarm will remain active until acknowledged by the operator. In conclusion, this architecture cannot be applied satisfactorily in B.

An alternative architecture is based on a top machine *Top* as single writer to the database, to which the device controllers and the monitoring console have only read access employing *SEES* or *USES* (Fig. 1 b). In this scenario, the device controllers cannot register alarms in the database directly. Instead, they have to return the corresponding information upon being called by *Top*, which in turn enters the alarms into the database. This becomes rather cumbersome if there are intermediate machines through which the information has to be passed or if the information does not have a constant length.

An additional problem becomes apparent when looking at the active alarms display operation of the monitoring console. This operation has to get a list of all active alarms.

This list not being constant in size, it cannot be returned by a single operation call. Instead, the elements have to be retrieved one by one –like the results of an SQL query in C. It is often simpler if the database maintains a set of already returned elements (See e.g. [6] for details of such a retrieval operation.) rather than requiring the search criteria together with a resume index to be passed with every call. However, if the retrieval operation updates a variable, it cannot be called by the monitoring console with read-only access. This problem could again be 'solved' by an undesirable underspecification.

Employing *SEES* in the device controllers we would be forced to specify properties relating devices and the database (e.g., the conveyor belt is only running if none of its alarms are active) in the *Top* machine rather than in the device controllers. With *USES* on the other hand, we would be allowed to reference variables of the database in the invariants of the device controllers, but would not be allowed to refine the latter leading either to a monolithic implementation of *Top* or a difficult to manage almost duplication of constructs.

In any case, the all including construct *Top* becomes the single focus of refinement without any direct support for architectural structure preserving refinement. Whereas it is definitely beneficial that B does not force the specification and implementation structures to be identical, the example shows that there are cases where more support for structure preserving refinement would be needed.

The problems of access restrictions can be overcome by merging all device controllers and the monitoring console into one big machine (Fig. 1 c). This, however, leads to a loss of modularity and, herewith, of information hiding, compositionality, reusability (e.g. multiple instantiation if we have several conveyor belts), and decomposition of proofs.

The same problems reoccur at the refinement and implementation level, where we are also restricted by the single writer approach and the limitations of shared variables visibility. The above problems are not limited to our specific example. Further motivation to analyze the reasons for the current restrictions and to suggest new mechanisms are, e.g., given by [22, Chapters 4, 5, 6, and 7].

3 Analysis of the Problem

In this section, we analyze the reasons for the single writer and shared variable visibility restrictions. In a nutshell, the restrictions are due to interference that would contradict compositionality and independent refinement by invalidating local proofs.

Consider a variation of the plant control system where alarms are set on the console. The conveyor belt simply adjusts its execution status based on whether there are active alarms in the system. The shared machine *Database2* contains a variable *activeAlarms*$\subseteq$*NAT*. The conveyor belt is specified as follows, using the keyword *UTILIZES* to indicate some sort of sharing access:

MACHINE ConveyorBelt2
UTILIZES Database2
VARIABLES running
INVARIANT running$\in$BOOL $\wedge$ (running=TRUE $\Rightarrow$ activeAlarms=$\emptyset$)

When the console sets an alarm it invalidates the invariant of *ConveyorBelt2* if *running* is *TRUE*. A construct with write access to a shared machine may invalidate any other accessor's invariant, if the latter references variables of the shared machine.

Such undesirable interferences cannot be ruled out with local proofs for any of the three machines *Database2*, *ConveyorBelt2*, or *Console2*. They require either a global approach or a modular approach with noninterference proofs like [20]. In both cases we would loose the benefits of independent refinement provided by a compositional theory [25].

Hence we have to choose between having multiple writers without the possibility to reference variables of the shared machine in any of the accessors' invariants or the current single writer paradigm. Not being able at to reference variables of the shared machine in any of the accessors' invariants is too restrictive, precluding the proving of many properties. For example, we cannot prove that the conveyor belt is only running if there are no active alarms because the variable *activeAlarms* of *Database2* is not visible in the invariant of *ConveyorBelt2*.

The same problems of destroying each other's invariants exist on the refinement and implementation levels. In addition to the local invariant, also the gluing invariant, expressing the data refinement relation, could be invalidated if we were to allow multiple writers [23].

On the positive side, we can note that multiple writers never invalidate the invariant of the shared machine as all modifications are done through operation calls.

4 Role-Based Access

To guarantee interference freedom among multiple accessors of a common machine, only the possible modifications to the shared variables are relevant. We define these effects in form of access *roles* as part of the shared machine. Accessing constructs declare which role(s) they play. The accessors guarantee to perform only modifications allowed by the declared role(s). In return, they can rely on the other accessors adhering to their roles. Let construct *A* accesses a shared machine in role R_1. If the other roles $R_2, \ldots, R_n$ maintain the invariant of *A*, then any accessor in role R_i ($i \in 2..n$) maintains the invariant of *A*. Thus, we can both specify and refine accessor *A* without knowing the other accessors' specifications or implementations.

Because a library machine might foresee multiple sharing scenarios and because a custom machine might be used in multiple instances with different sharing, we allow the definition of multiple *contracts* with different roles.

4.1 Role Specifications

We illustrate the concept on our plant control system. This time, the conveyor belt creates the alarms reacting to sensors and an emergency stop button. The monitoring console is used to deactivate alarms. *Database3* defines a contract *SingleDevice* with two roles *Creator* and *Controller*, intended to capture the accesses by *ConveyorBelt3* and *Console3* respectively. An instance of *Database3* with contract *SingleDevice* can have at most two accessors, one for each role.

```
MACHINE Database3
CONTRACTS
  SingleDevice ≙
    Creator = ANY type WHERE type∈NAT THEN NewAlarm(type) END,
    Controller = ANY aa WHERE aa∈activeAlarmsTHEN ResetAlarm(aa) END
VARIABLES alarms, activeAlarms, alarmType
INVARIANT alarms⊆NAT ∧ activeAlarms⊆alarms ∧ alarmType∈alarms→NAT
INITIALISATION alarms, activeAlarms, alarmType:=∅, ∅, ∅
OPERATIONS
  aa ← NewAlarm(type) ≙
    PRE type∈NAT THEN
      ANY nn WHERE nn∈NAT-alarms THEN
        aa, alarms:=nn, alarms∪{nn} ||
        activeAlarms, alarmType(nn):=activeAlarms∪{nn}, type
      END
    END;
  ResetAlarm(aa) ≙ PRE aa∈activeAlarms
    THEN activeAlarms:=activeAlarms-{aa} END;
  nof ← NofActiveAlarms ≙ nof:=card(activeAlarms);
  ...
END
```

We specify the set of alarms as a subset of *NAT* and the active alarms as a subset of all alarms. The attribute *alarmType* is a functions from *alarms* to *NAT*. More on this approach of mapping records/classes to B, including proper treatments of finiteness, can be found in [18, 6].

4.2 Accesses

The machine *ConveyorBelt3* declares that it accesses the *Database3* as *Creator* in a *SingleDevice* contract. We use a '*!*' as separator of the qualified identifier because the dot is reserved for possible renaming.

```
MACHINE ConveyorBelt3
ACCESSES Database3!SingleDevice AS Creator
VARIABLES running
INVARIANT running∈BOOL ∧ (running=TRUE ⇒ activeAlarms=∅)
INITIALISATION running:=TRUE
OPERATIONS
  rr ← Run ≙
    CHOICE
      ANY type WHERE type∈0..8
        THEN NewAlarm(type) || running, rr:=FALSE, FALSE END
    OR running, rr:=bool(activeAlarms=∅), bool(activeAlarms=∅)
    END;
  EmergencyStop ≙ BEGIN NewAlarm(9) || running:=FALSE END
END
```

The operations *Run* and *EmergencyStop* have to act as refinements of *Creator* ‖ *skip* on the state space of the *Database*, which is clearly the case. This also implies, that inquiry operations can be freely called by any accessor.

Furthermore, we need to show that another construct, accessing *Database3* in the second role *Controller* cannot invalidate the invariant of *ConveyorBelt3*. To this aim, we show that the role specification *Controller* executed like an operation on the combined state space of *Database3* and *ConveyorBelt3* maintains the latter's invariant. This is the case, because *Controller* can only deactivate alarms. Deactivation is unproblematic because the second conjunct of the invariant of *ConveyorBelt3* is an implication and not an equality.

4.3 Refining and Implementing Accesses

Refinements and the implementation of *ConveyorBelt3* have to make the same changes to the variables of the shared machine *Database3*.

We assume *Motor3* to be a machine controlling the power of the motor and *Sensor3* a sensor that is activated if a load on the conveyor belt is about to fall off the edge.

```
IMPLEMENTATION ConveyorBelt3′ REFINES ConveyorBelt3
ACCESSES Database3!SingleDevice AS Creator
IMPORTS M.Motor3, S.Sensor3
INVARIANT running=M.on
OPERATIONS
  rr ← Run ≙
    VAR ss, nof IN
      ss ← S.ReadSensor;
      IF ss=TRUE THEN M.ShutOff; NewAlarm(0)
      ELSE nof ← NofActiveAlarms; IF nof=0 THEN M.TurnOn END
      END
    END;
  EmergencyStop ≙ BEGIN M.ShutOff; NewAlarm(9) END
END
```

Instead of accessing the database itself, the implementation *ConveyorBelt3′* could also import another machine that accesses the database and performs the changes.

4.4 Instantiation

In the existing composition mechanisms, the single writer also instantiates the machine parameters of the utilized machine. In our new mechanism, instantiation is separate from access using the *INSTANTIATES* clause, which specifies the machine, the contract, and the values of the parameters, if any. For example, we might have an implementation *Main3′*, which imports the accessors and instantiates the shared database:

```
IMPLEMENTATION Main3′ REFINES Main3
INSTANTIATES Database3!SingleDevice
IMPORTS ConveyorBelt3, Console3
```

Every accessed copy of a shared machine must be instantiated exactly once in an implementation, naming the same contract as all accessors. Renaming can be performed in the *ACCESSES* and *INSTANTIATES* clauses, thus allowing multiple instances with possibly different contracts. The renaming of the construct containing the *INSTANTIATES* clause determines the number of instances.

5 Further Aspects of Role-Based Access

5.1 Replicated Roles

In the previous section we have used role-based access for a plant control with a single device. In reality, we have many devices, which all have almost identical role specifications. Rather than requiring textual duplication, we introduce a replication mechanism over a constant set. Thus, we can define the role *Creator* of *Database4* as follows:

MACHINE Database4
CONTRACTS
MultipleDevices $\widehat{=}$
Creator(no$\in$0..19) $=$
ANY type **WHERE** type$\in$10$\times$no..10$\times$no+9 **THEN** NewAlarm(type) **END**,

This example definition allows 20 accessors in the role of creators, one for each value between 0 and 19. The replicator *no* may be used inside the scope of the role definition like a constant. A construct that accesses a shared machine in a replicated role has to indicate its replication value. The conveyor belt could be defined as:

MACHINE ConveyorBelt4
ACCESSES Database4!MultipleDevices **AS** Creator(0)

For the non-interference proofs the other replicated roles have to be considered like different role specifications. In the example, we would have to prove that *Creator(nn)* for *nn*$\in$*1..19* maintains the invariant of *ConveyorBelt4*. This is the case if we adapt the invariant of *ConveyorBelt3* as below and adjusting the *Run* operation correspondingly.

INVARIANT
running$\in$BOOL $\wedge$ (running=TRUE $\Rightarrow$ activeAlarms$\cap$(alarmType^{-1}[0..9])=$\emptyset$)

5.2 Form of Role Specifications

As shown in the examples, role specifications take the format of normal B operations. Traditionally, rely/guarantee conditions are expressed as predicates over the current and the next state of variables. However, we feel that operation-like specifications are more in line with B.

As a guiding principle, we allow the same statements as in operations of a refinement that includes the accessed machine. Thus, multiple substitutions, sequencing, and nondeterministic choice are all allowed, but loops are not. Variables can be read directly, but modified through operation calls only. To gain sufficient expressiveness, either loops or direct modifications should be made legal.

We do not have to prove that operation calls in role specifications satisfy the preconditions of the called operations, because we perform these proofs for the actual accessors. As an engineering aid, the tools should nevertheless support conformance proofs for role specifications. Precondition violating role specifications do not give the accessor more freedom, but make the non-interference proofs for other accessors more difficult.

The role specifications, like any module interfaces, should be very simple compared to the code of the actual accessors. Hence, roles are described like operations, rather than full machines with variables that maintain their values between calls. Such specifications would require full-blown construct refinement with gluing invariants between role and local variables in accessors and also more complex non-interference proofs. The simple format suffices in most cases and is, combined with some coding tricks modifying the operation specifications, as general as full machines.

The relative simplicity of the role specifications compared to the code of the actual accessors reduces the complexity of the non-interference proofs. The simplification of the non-interference proofs for all other accessors on all refinement levels by far outweighs the additional burden of the single role adherence proof.

An overly weak role specification makes it easy to prove role adherence, but impossible to guarantee non-interference for other accessors. An overly strong specification causes the opposite problem. Writing the role specifications is a design step, like any other definition of module interaction.

5.3 Adherence to Role Specifications

An operation of an accessor adheres to a role specification if it either acts as a refinement of the specification or as *skip* on the state space of the accessed machine. We add *skip* as an implicit choice to every role specification. This corresponds to the guarantee condition being reflexive [15], respectively the stuttering transition being built into the semantics [3, 17].

Whether an operation O that refines a role specification R is called multiple times or whether O acts as a refinement of $R\hat{}$ $(=skip[\!]R[\!](R;R)[\!]\ldots)$ has the same effect for the other accessors. This corresponds to the guarantee condition being transitive [15], respectively mumbling being built into the semantics. Explicitly allowing mumbling makes the proofs more difficult and can –provided we allow direct write access or loops in role specifications– always be replaced with a weaker role specification. For simplicity, we do not consider mumbling in this paper.

In the initialization of the accessors we only allow inquiry operations of the accessed machine to be called. Otherwise we would have to define an order in which the accessors are initialized and could not assume the shared machine to be in its initial state when the accessors are initialized. The initializations acting as *skip* on the accessed machine, they automatically adhere to the role specifications.

5.4 Sharing Structure

Sharing is used to get multiple access paths to the same data. In the presence of independent refinement, we need some structural restrictions to control aliasing. In this section, we give two examples of what could go wrong without such structural restrictions. A full account of the restriction is given in Sect. 6.4.

We adopt the following notation in figures: The primed constructs are the implementations refining the unprimed machines. Multiple instances of an accessed machine –if present– are graphically indicated by duplication to make collaborations clear. We

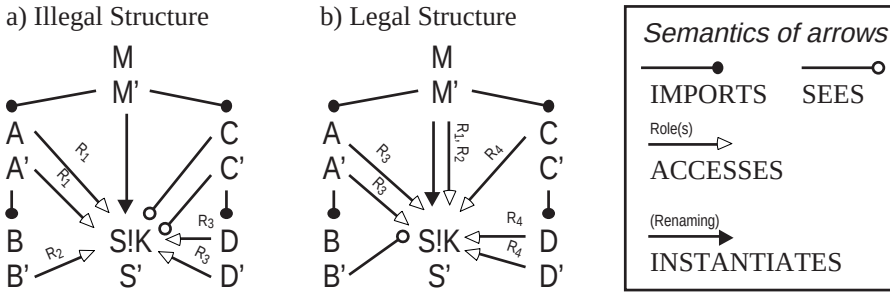

Fig. 2. Illustration of Legal and Illegal Composition Graphs

append the name of the actual contract to the name of the accessed machine. The access arrows are adorned by the role(s) and possibly the replication values, the instantation arrow by the possible renaming. In a slight abuse of notation, it would also be possible to visualize roles as UML style interfaces (circles) attached to the shared machine. However, our notion of roles and that of UML interfaces is not identical because our roles contain guarantee conditions rather than the signatures of callable operations.

Consider a machine A that accesses a shared machine S (left branch of Fig. 2 a). The implementation A' also accesses S and furthermore imports B. Machine B does not access S, but the implementation B' does. Even if we locally prove that the operations of A' act as a refinements of the operations of A, this property might not hold in the complete system. A' may call operations of B and, thereby, unknowingly modify S. This might lead to S being modified differently than specified in A, B' observing S in a state where the gluing invariant of B' does not hold, and A' violating preconditions of operations of S. This problem is due to A' accessing S both traceably and untraceably. The problem is not bound to B only accessing S in the implementation. An invisible access could also be created if B' would not access S directly, but import a machine E that accesses S.

Without constraints on the composition graph, also the interaction between the old and the new composition mechanism can lead to problems. A seeing construct C' (right branch of Fig. 2 a) assume that the state of the seen machine S does not change during the execution of an operation of C'. To enforce this, the seeing construct C' can only call inquiry operations. In proofs of C', no substitutions are made on the state of the seen machine S. If C' could indirectly modify S, global correctness would be invalidated. This could, for example, happen if the seeing machine imports a third machine D, the implementation of which accesses S. Thus, we need restrictions on the structure of the development to ban such architectures.

5.5 Emulating the Existing Composition Mechanisms

The existing composition primitives *IMPORTS* and *SEES* can be emulated using *ACCESSES* and *INSTANTIATES* as follows: A contract permitting a single writer with full access rights and an infinite number of readers with a *skip* role specification is added

to the shared construct. Then, *IMPORTS* can be replaced with an access in the writer role and an instantiation. The *SEES* clauses are replaced with accesses in the replicated reader role. Because the existing mechanisms *IMPORTS* and *SEES* capture a frequent special case and because abolishing them would require more complicated global restrictions based not only on the structure but also the semantics of roles, it makes sense to have all mechanisms at our disposition.

Promotion of operations (turning operations of a utilized machine into proper operations of the utilizing construct) would only be possible in combination with *ACCESSES* in trivial cases where the other accessors do not make any observable modifications (e.g., for the single writer in the above emulation contract). Furthermore, promotion of operations is much less important with *ACCESSES*, because the latter provides for multiple writers. Therefore, we do not consider the promotion of operations from accessed machines, but rather count promotion as a further reason to also keep the existing import mechanism.

The *USES* mechanism cannot be emulated because it dictates that the used machine be included into another machine whereas an accessed machine cannot be included. *INCLUDES*, being a copying rather than a sharing mechanism, cannot be emulated with *ACCESSES*.

6 Formal Definitions

6.1 Syntax

We give the following extended syntax definitions [2, p 715ff] for machines, refinements, and implementations:

MACHINE
Machine_Header
CONSTRAINTS
Predicate
CONTRACTS
Contract_List
ACCESSES
Access_List
INSTANTIATES
Inst_List
USES
...

REFINEMENT
Machine_Header
REFINES
Id_List
ACCESSES
Access_List
INSTANTIATES
Inst_List
SETS
...

IMPLEMENTATION
Machine_Header
...
VALUES
Predicate
ACCESSES
Access_List
INSTANTIATES
Inst_List
IMPORTS
...

Contract_List, *Access_List*, and *Inst_List* are defined as follows:

Syntactic Category	Definition
Contract_List	Contract; Contract_List Contract
Contract	Contract_Name $\widehat{=}$ Role_List

Syntactic Category	Definition
Role_List	Role, Role_List Role
Role	Role_Name = Statement Role_Name(Replicator ∈ Set) = Statement
Access_List	Access; Access_List Access
Access	Machine_Name!Contract_Name **AS** Acc_Role_List Renamed_Name.Machine_Name!Contract_Name **AS** Acc_Role_List
Acc_Role_List	Acc_Role, Acc_Role_List Acc_Role
Acc_Role	Role_Name Role_Name(Simple_Term)
Inst_List	Inst, Inst_List Inst
Inst	Machine_Name!Contract_Name Machine_Name!Contract_Name(Expression_List) Renamed_Name.Machine_Name!Contract_Name Renamed_Name.Machine_Name!Contract_Name(Expression_List)

Contract_Name, *Role_Name*, *Replicator*, *Machine_Name*, and *Renamed_Name* all stand for *Identifier*. The form of the role specification is discussed in detail in Sect. 5.2.

6.2 Proof Obligations

We give the proof obligations for machines and implementations containing an *ACCESSES* clause. The rules for refinements are analogous. As noted in Sect. 5.2, the *CONTRACTS* clause does not give rise to any proof obligations. We leave out sets, constants and assertions as the respective proof obligations are unchanged. Figure 3 gives an overview of the proof obligations.

MACHINE $M_s(P_s)$
CONSTRAINTS C_s
CONTRACTS
$K \mathrel{\widehat{=}}$
$R_1 = F_1$,
$R_2 = F_2$
VARIABLES X_s
INVARIANT I_s
INITIALISATION U_s
OPERATIONS
$u_s \leftarrow O_s(w_s) \mathrel{\widehat{=}}$
PRE Q_s **THEN** V_s **END**
END

MACHINE $M_1(P_1)$
CONSTRAINTS C_1
ACCESSES
M_s!K **AS** R_1
VARIABLES X_1
INVARIANT I_1
INITIALISATION U_1
OPERATIONS
$u_1 \leftarrow O_1(w_1) \mathrel{\widehat{=}}$
PRE Q_1 **THEN** V_1 **END**
END

IMPLEMENTATION
$M'_1(P_1)$
REFINES M_1
ACCESSES
M_s!K **AS** R_1
CONCRETE_VARS
X'_1
INVARIANT I'_1
INITIALISATION U'_1
OPERATIONS
$u_1 \leftarrow O_1(w_1) \mathrel{\widehat{=}}$
BEGIN V'_1 **END**
END

We use the abbreviations A_1 for $P_1 \in \mathcal{P}_1(\mathrm{INT})$ and A_s for $P_s \in \mathcal{P}_1(\mathrm{INT})$. Occurrences of O_s in U_1, V_1, U'_1, V'_1, F_1, and F_2 should be replaced by V_s with the parameters substituted accordingly [2, p 314ff]. As in [2] we do not make this substitution explicit in the proof obligations.

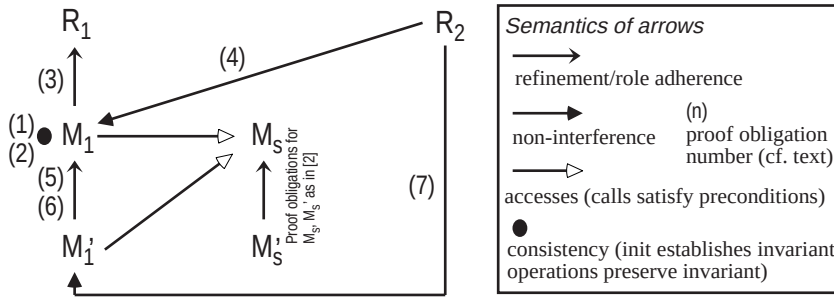

Fig. 3. Proof Obligations for an Accessing Machine and Implementation

Machine M_1 The first proof obligation of M_1 states that the initialization must establish the invariant. The role of the accessed machine is similar to the one of an included machine, except that its parameters are not actualized [2, p 331ff].

$$A_1 \wedge C_1 \wedge A_s \wedge C_s \Rightarrow [U_s][U_1]I_1 \tag{1}$$

The next obligation concerns the preservation of the invariant of the accessing machine by its operations:

$$A_1 \wedge C_1 \wedge I_1 \wedge Q_1 \wedge A_s \wedge C_s \wedge I_s \Rightarrow [V_1]I_1 \tag{2}$$

The third obligation states that the operations of the accessor must conform to the declared role. Note that there is no corresponding obligation for the initialization because the latter may not call modification operations of the accessed machine. Because both the role specification F_s and the operation O_1 operate on X_s, renaming must be performed. Let $\hat{X}_s$ be a fresh set of variables, then we get

$$A_1 \wedge C_1 \wedge I_1 \wedge Q_1 \wedge A_s \wedge C_s \wedge I_s \wedge \hat{X}_s{=}X_s \Rightarrow [[X_s{:=}\hat{X}_s]V_1]\neg[F_1 \,\|\, \textbf{skip}]\neg(\hat{X}_s{=}X_s) \tag{3}$$

If a construct accesses a machine in multiple roles, its operations have to conform to the nondeterministic choice of the two roles. Thus, if M_1 were to access M_s as R_1 and R_2, then F_1 would have to be replaced by $F_1 \| F_2$.

The fourth obligation concerns the interference freedom by all other roles, which in our case is only R_2.

$$A_1 \wedge C_1 \wedge I_1 \wedge A_s \wedge C_s \wedge I_s \Rightarrow [F_2]I_1 \tag{4}$$

For replicated roles, we have to prove non-interference for all replication values except for the one of the accessor in question. Let us assume the following replications $R_1(g_1 \in G_1)$ and $R_2(g_2 \in G_2)$ and let M_1 access M_s as $R_1(h_1)$. Then we get the following three obligations:

$$h_1 \in G_1 \tag{4'}$$
$$A_1 \wedge C_1 \wedge I_1 \wedge A_s \wedge C_s \wedge I_s \wedge g_1 \in G_1\text{-}\{h_1\} \Rightarrow [F_1]I_1$$
$$A_1 \wedge C_1 \wedge I_1 \wedge A_s \wedge C_s \wedge I_s \wedge g_2 \in G_2 \Rightarrow [F_2]I_1$$

Replication not only avoids duplication of role specifications, it also leads to a reduction of the overal proof burden by combining many similar non-interference obligations.

The proof obligations for operation calls (satisfy precondition) are unchanged.

Implementation M'_1 For the implementation M'_1 we have 3 proof obligations. The first two proof obligations concerning initialization and operation refinement are similar to those of an implementation that imports another machine [2, p 597ff].

$$A_1 \wedge C_1 \wedge A_s \wedge C_s \Rightarrow [U_s][U'_1]\neg\, [U_1]\neg I'_1 \tag{5}$$

The second proof obligation is for the operation refinement. The 1-to-1 data refinement of the shared variables is explicit in this obligation *($\hat{X}_s$=X_s)*.

$$\begin{aligned} &A_1 \wedge C_1 \wedge I_1 \wedge I'_1 \wedge Q_1 \wedge A_s \wedge C_s \wedge I_s \wedge \hat{X}_s{=}X_s \Rightarrow \\ &\quad [[u_1{:=}\hat{u}_1][X_s{:=}\hat{X}_s]V'_1]\neg[V_1]\neg([X_s{:=}\hat{X}_s]I'_1 \wedge \hat{u}_1{=}u_1 \wedge \hat{X}_s{=}X_s) \end{aligned} \tag{6}$$

For sharing in the implementation only, we have to prove adherence rather than 1-to-1 data refinement in the implementation. The third and last obligation concerns the interference freedom. As noted above for machines, it should be replicated if some of the roles are.

$$A_1 \wedge C_1 \wedge I_1 \wedge I'_1 \wedge A_s \wedge C_s \wedge I_s \Rightarrow [F_2]I'_1 \tag{7}$$

If M or M'_1 also instantiates M_s, say P_s with N_s, then A_s can be replaced by the stronger predicate P_s=N_s in the above proof obligations. In this case we have the additional proof obligation that the actual parameters satisfy the constraints, as for *INCLUDES* and *IMPORTS*.

6.3 Visibility Rules

For brevity, we only summarizes some key aspects of the visibility rules here. In the *CONTRACTS* clause we allow only read access to variables. A construct's own sets and constants as well as those of seen machines are allowed as parameters of instantiations. To prevent cyclic dependencies, sets and constants of included, used, imported, and accessed machines are, on the other hand, not visible in the *INSTANTIATES* clause.

Only a construct's own sets and constants and those of seen machines, but not those of imported machines may be used as parameters of instantiations. In their initializations, accessors can call only inquiry operations of an accessed construct.

If a construct *A* only instantiates, but does not access *B*, then none of the objects of *B* are visible in *A*. Like *SEES*, but unlike *INCLUDES*, *ACCESSES* is not transitive. If machine *A* includes, uses, sees, imports, or accesses *B* and *B* accesses *C*, then the objects of *C* are not visible in *A*. It is, however, possible that *A* also accesses *C* (Fig. 2 b).

6.4 Well-Formedness of the Composition Graph

The well formedness criteria for the composition graph concerning *ACCESSES* to guarantee global correctness are presented below. They are simple enough to be checked automatically.

Similar checks are already performed for the existing composition mechanisms [2, 21]. For simplicity, we talk about 'accessed machines' instead of renamed instances thereof.

The following conditions can be verified by the type checker on a per-construct base:

1. If a machine, a refinement, or an included machine thereof accesses a machine M_s as R of contract K then this construct's implementation must either access M_s as R of contract K or import exactly one machine that contains such an access.
2. If a machine, a refinement, or an included machine thereof accesses a machine M_s as $R_1, \ldots, R_i$ of contract K then this construct's implementation may not access M_s in any other roles nor import a machine accessing M_s in any other roles. (The proof obligation for operation refinement would not allow modifications not covered by $R_1, \ldots, R_i$ anyhow.)
3. A construct and one of its included machines cannot access the same machine in the same role.
4. If a machine, a refinement, or an included machine thereof contains an instantiation, then all further refinements and the implementation must either contain the same instantiation with the same parameters or include/import without renaming a machine containing such an instantiation.

The following conditions must be checked globally for complete projects:

1. Every accessed machine is instantiated exactly once in an implementation.
2. Every shared machine is accessed at most once in each role, respectively for each replication value, by an implementation
3. All accesses and the instantiation of a machine are for the same contract.
4. An accessed machine cannot be included or imported. This also implies that neither a used nor a using machine can be accessed.
5. A seen machine must either be instantiated or imported.

To present the remaining architectural condition, we extend the notation of [21]. The relational notation is as in B: '$^+$' denotes the transitive non-reflexive closure, '*' the transitive and reflexive closure, and ';' composition.

1. M_1 *sees* M_2 iff the implementation of M_1 sees the machine M_2.
2. M_1 *m_sees* M_2 iff machine M_1 sees machine M_2.
3. M_1 *imports* M_2 iff the implementation of M_1 imports the machine M_2.
4. M_1 *accesses* M_2 iff the implementation of M_1 accesses the machine M_2.
5. M_1 *depends_on* M_2 iff the implementation of M_1 is built utilizing M_2: *depends_on* $\widehat{=}$ *(sees*$\cup$*imports*$\cup$*accesses)*$^+$.
6. M_1 *can_alter* M_2 iff the implementation of M_1 can alter the variables of M_2: *can_alter* $\widehat{=}$ *depends_on***; (imports*$\cup$*accesses).*

7. M_1 *any_accesses* M_2 iff M_1, one of its refinements, or its implementation accesses the machine M_2.
8. M_1 *(imp_acc* M_s*)* M_2 iff the implementation of M_1 imports the machine M_2 and M_2 accesses M_s.
9. M_1 *traceably_accesses* M_2 iff M_1 accesses M_2 through a chain of imports, in which all machines access M_2:
   M_1 *traceably_accesses* $M_2 \mathrel{\widehat{=}} M_1$ *(imp_acc* M_2*)***; accesses* M_2.
10. M_1 *untraceably_accesses* M_2 iff M_1 indirectly accesses M_2 in a way other than an imports chain, in which all machines access M_2:
    untraceably_accesses $\mathrel{\widehat{=}}$ *(depends_on; accesses)-traceably_accesses*.
11. M_1 *instantiates* M_2 iff the implementation of M_1 instantiates the machine M_2.
12. M_1 *references* M_2 iff the implementation of M_1 references the machine M_2: *references* $\mathrel{\widehat{=}}$ *(sees*$\cup$*imports*$\cup$*accesses*$\cup$*instantiates)*$^+$.
13. *id* is the identity relation.

The composition graph must then satisfy the following condition:

$$((\text{sees} \cup \text{imports} \cup \text{accesses}); \text{can\_alter}) \cap (((\text{imports} \cup \text{accesses}); \text{m\_sees}^+) \cup (\text{sees}; \text{m\_sees}^*)) = \emptyset \wedge \quad \text{(i)}$$

$$\text{any\_accesses} \cap \text{untraceably\_accesses} = \emptyset \wedge \quad \text{(ii)}$$

$$\text{references} \cap \text{id} = \emptyset \quad \text{(iii)}$$

The first conjunct states that a seen machine must not be modified. The second conjunct asserts that no construct accesses the same machine directly and untraceably. The third conjunct excludes cyclic dependencies.

The right branch of Fig. 2 a) violates the first conjunct of the above condition, the left branch violates the second conjunct. M' not accessing S has nothing to do with the violations; the corresponding access in the left figure is just shown as an additional option.

7 Soundness

In this section we give a partial proof of the soundness of our new shared access mechanism. We syntactically merge a shared machine and all its accessing machines into a new machine and the implementation of the shared machine along with the implementations of the accessors into a new implementation. Then we show that all the proof obligations of these two constructs, which do not contain the new mechanism, are implied by the obligations of the individual constructs. Namely, the invariant of the merged machine holds and the implementation is a correct refinement.

Because we have substitutions of both V_s and V'_s for O_s, we have to indicate which body is used. We write $[O_s \backslash V_s]$ for this extended substitution which includes the parameters, e.g., $[O_s \backslash V_s]$ $(a \leftarrow O_s(b))$ equals $[u_s, w_s{:=}a, b]V_s$ if u_s is the output and w_s the input parameter of O_s. We assume here that operations are not recursive.

Let M_s, M_1, and M'_1 be as in Sect. 6.2. Furthermore, let M_2 and M'_2 be like M_1 and M'_1 respectively, but with index '2'. With M'_S as in Fig. 4, we get the two merged constructs M and M' (Fig. 4). Note that V'_s gets substituted for O_s in the implementation M'.

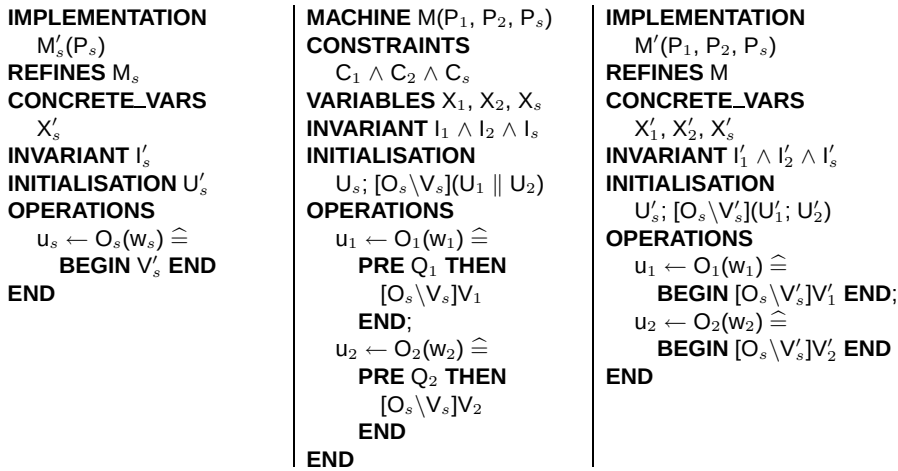

Fig. 4. Flattened Constructs

Theorem 1. *If all proof obligations of* M_s, M'_s, M_1, M'_1, M_2, *and* M'_2 *are true ([2, p 763ff], Sect. 6.2), then all proof obligations of* M *and* M′ *hold.*

Several soundness proofs of the rely/guarantee method for shared variable systems have been given in the literature for different formalisms [24, 27, 1, 12]. The proof of this theorem is very similar.

8 Summary

8.1 Related Work

The use of assumptions and commitments to achieve compositionality in program verification was first proposed by Francez and Pnueli [11]. Jones introduced rely/guarantee conditions as a method for top-down program development [15]. Ketil Stølen has added wait-conditions to handle synchronization and auxiliary variables to increase expressiveness [24]. Jones himself applied the idea to object-oriented systems [16]. Rely/guarantee specifications have also been incorporated into temporal logic-based formalisms, thereby also capturing certain liveness properties: Collete added them to UNITY [7] and Abadi and Lamport to TLA [1]. Misra and Chandy have first used assumption/commitment specifications for message passing systems [19]. A unifying overview of shared variable and message passing assumption/commitment specifications is given by [26].

Neither VDM nor Z have an equally powerful modularization mechanism as B, although some constructions have been suggested [9, 13]. RAISE, Cogito, and other related formalisms provide different forms of modularization. However, we are not aware of any compositional symmetric shared access mechanism comparable to ours.

Both Jones [15] and Stølen [24] combined rely/guarantee specifications with a VDM like logic and syntax. However, their aim was to reason about concurrent programs only and they have not investigated rely/guarantee specifications in VDM for modular sharing. Whereas existing work has mostly focused on the use of assumption/commitment specifications for concurrent system, this paper has applied them to achieve compositionality in sequential systems with shared components.

Role-based contracts for different forms of collaborations have been proposed, e.g., by Helm et al for object-oriented systems [14] and by Francez and Forman for interacting processes [10]. Role-based specifications expressing rely/guarantee conditions as part of the shared construct are believed to be new. Traditionally, a rely/guarantee pair is part of each component to be composed. Centralization of all rely/guarantee specifications is possible in our case because only a single component is shared, whereas most other approaches handle mutual sharing. Our benefit is that all proofs for an accessor can be performed without knowing the other accessors.

Pioneering work in explaining the existing B composition mechanisms and their interplay with refinement has been done by Bert, Potet, and Rouzaud [5, 21].

8.2 Conclusions

We have extended the B method with a compositional symmetric shared access mechanism that overcomes the limitations of the single-writer restriction and the limited visibility of shared variables of the existing mechanisms. Based on rely/guarantee conditions expressed as accessor roles of the shared construct, the new mechanism is compositional, providing for independent refinement without the need to know the other accessors. The abstraction of possible modifications into compact role specifications simplifies the non-interference proofs. The new mechanism provides for flexible sharing on all levels; applications with sharing requirements can be specified, refined, and implemented without loss of modularity or underspecification as has been the case with the existing mechanisms. Uniform applicability in all constructs, replicated roles, multiple contracts, and good integration with existing composition mechanism add to the flexibility of the new mechanism.

For the new mechanism, we have given formal definitions of the syntax, the proof obligations, the visibility rules, and the restrictions on the composition graph. A partial soundness proof completes the paper.

Acknowledgments. Marina Waldén and Emil Sekerinski provided detailed comments on earlier drafts. We would also like to thank Wolfgang Weck for a number of fruitful discussions on the topic. The referees' comments are gratefully acknowledged.

References

[1] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[2] J. R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] R. Back and J. von Wright. Trace refinement of action systems. In *CONCUR 94*, pages 367–384. LNCS 836, Springer Verlag, 1994.
[4] J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990.
[5] Didier Bert, Marie-Laure Potet, and Yann Rouzaud. A study on components and assembly primitives in B. In *Proceedings of the first B conference*, pages 47–62, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, 1996. IRIN Institut de recherche en informatique de Nantes.
[6] Martin Büchi. The B Bank. In Emil Sekerinski and Kaisa Sere, editors, *Program Development by Refinement: Case Studies Using the B Method*, chapter 4, pages 115–180. Springer Verlag, 1998. http://www.abo.fi/~mbuechi/publications/BBook.html.
[7] Pierre Collette. Application of the composition principle to UNITY-like specifications. In *Proceedings of TAPSOFT 93*, pages 230–242. LNCS 668, Springer Verlag, 1993.
[8] Willem-Paul de Roever. The quest for compositionality—a survey of assertion-based proof systems for concurrent programs, part I: Concurrency based on shared variables. In F.J. Neuhold and G. Chroust, editors, *Proceedings of the IFIP Working Conference "The role of abstract models in computer science"*, pages 181–205. North-Holland, 1985.
[9] J.S. Fitzgerald and C. B. Jones. Modularizing the formal description of a database system. In *VDM'90: VDM and Z – Formal Methods in Software Development*, pages 189–210. LNCS 428, Springer Verlag, 1990.
[10] N. Francez and I. Forman. *Interacting Processes: A Multiparty Approach to Coordinated Distributed Programming*. ACM Press, 1996.
[11] Nissim Francez and Amir Pnueli. A proof method for cyclic programs. *Acta Informatica*, 9:133–157, 1978.
[12] Peter Grønning, Thomas Qvist Nielsen, and Hans Henrik Løvengreen. Refinement and composition of transition-based rely-guarantee specifications with auxiliary variables. In *Proceedings of the 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 332–348. LNCS 472, Springer Verlag, 1990.
[13] I. J. Hayes and L. P. Wildman. Towards libraries for Z. In J. P. Bowen and J. E. Nicholls, editors, *Z User Workshop: Proceedings of the Seventh Annual Z User Meeting*, Workshops in Computing. Springer Verlag, 1993.
[14] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of OOPSLA/ECOOP '90*, pages 169–180, 1990.
[15] Cliff B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North Holland, 1983.
[16] Cliff B. Jones. Accomodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.
[17] Leslie Lamport. The temporal logic of actions. *ACM Transactions of Programming Languages and Systems*, 16(3):872–923, 1994.
[18] Kevin Lano. Integrating formal and structured methods in object-oriented system development. In S.J. Goldsack and S.J.H. Kent, editors, *Formal Methods and Object Technology*. Springer Verlag, 1996.
[19] J. Misra and M. Chandy. Proofs of networks of processes. *IEEE Software Engineering*, 7(4):417–426, 1981.
[20] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
[21] Marie-Laure Potet and Yann Rouzaud. Composition and refinement in the B-method. In *Proceedings of the second B conference*, pages 46–65. LNCS 1393, Springer Verlag, 1998.
[22] Emil Sekerinski and Kaisa Sere, editors. *Program Develoment by Refinement: Case Studies Using the B Method*. FACIT. Springer Verlag, 1998.

[23] Kaisa Sere and Marina Waldén. Data refinement of remote procedures. In *Proceedings of TACS 97*, pages 267–294. LNCS 1281, Springer Verlag, 1997.

[24] Ketil Stølen. *Development of Parallel Programs on Shared Data-Structures*. PhD thesis, University of Manchester, 1990. Available as technical report UMCS-91-1-1.

[25] Qiwen Xu. On compositionality in refining concurrent systems. In J. He, J. Cooke, and P. Wallis, editors, *Proceedings of the BCS FACS 7th Refinement Workshop*. Electronic Workshops in Computing, Springer Verlag, 1996.

[26] Qiwen Xu, Antonio Cau, and Pierre Collette. On unifying assumption-commitment style proof rules for concurrency. In *Proceedings of CONCUR 94*, pages 267–282. LNCS 836, Springer Verlag, 1994.

[27] Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.