Technical note. On Sequentiality and Concurrency, 28 March 2013
(Edited 20 April 2014; added initial remarks)
Stefan Hallerstede,
Aarhus University Department of Engineering, Aarhus, Denmark,
stefan.hallerstede@wanadoo.fr

On this note. This note collects ideas and an attempt for modelling of concurrent programs starting from Event-B and incorporating concepts for dealing with structure (in the form of proof outlines) and with composition (in the form of rely-guarantee). This work is ongoing and unfinished. The purpose of this note is stock keeping. Some of the better ideas were born while working on this and some of the bad ideas would disappear without trace. And this would be a shame because the motivation for the next steps comes directly from the failures of the current state of affairs. This note mostly serves for future reference and does not provide a solution to the stated modelling problem. The following paragraph describes the main ideas behind the work relevant for this note.

Abstract. Based on methods for program verification by refinement such as the B-Method or VDM we propose an approach for verifying programs composed of sequential and concurrent parts. Similar to the approach to modelling and proving taken in Event-B we focus on the support for incremental development and ease of proof. We achieve this by following the style of proof obligations used in Event-B. Mixing sequentiality and concurrency requires some structuring in the modelling notation. The main challenge of our approach is permitting such structuring while keeping proof as easy as possible. To achieve this we pragmatic with respect to the proof techniques employed borrowing from Hoare-style program verification and rely-guarantee reasoning. The approach is suitable for reasoning about abstract models of programs where refinement is used for structuring a complex proof. It does not describe a development method by step-wise refinement but an approach to fill in gaps in a complex proof. Refinement is used to produce proofs comparable to those obtained by Hoare-style verification but with flexibility obtained by refinement techniques.

Notation. Having carried out the experiment on the method and notation, we find that outlines seem to be an efficient tool for the purpose. In particular, the "dot-notation" are not appropriate for the propoesed reasoning. It is redundant and difficult to read. The intention was to render the underlying method more visible, the opposite of what has been achieved. Similarly, the indication of the refinement relationships for commands in the proof outlines in the form "$C\backslash A$" with abstract command A and concrete command C id difficult to read and it limits the kinds of refinement relationships that can be specified. This has lead to the requirement that some commands must be refined by skip which is very artifical and difficult to motivate. Protocols are too restricting. We need a more "loose" method for dealing with this. Specifying pairs of places and sets of commands may work. These also make immediately sense "locally" in proof outlines not considering composition. In earlier version we used places with tolerances. This turned out to complicate composition and introduce unnecessary choices during modelling and proof that would not contribute to the actual correctness proof. The attachment to commands removed this problem. In a discussion with Cliff Jones (using rely-guarantee terminology) prior to writing this he pointed out that preferred to attach such a concept to commands. In the end we followed his advice. Finally, in the note we use predicative notation to express proof obligations requiring renaming and some side-conditions on variable naming. This complicates the presentation and mixes syntactic problems with methodological ones. Since this was written we have used set-theory and treated syntax apart. This leads to much more readable and concise formulas and makes the presentation insensitive to naming issues. The program notation is not suitable for modelling but was considered appropiate for the proposed refinement method. It was an attempt to preserve some of the usual constructs. The concrete syntax has evolved from an earlier version publishes ABZ 2010. Even if the usual syntax would be introduced the mismatch between notation and method would remain. Effectively, we have two if-statements, one by itself and one to occur just after a loop. All in all, it is too complicated to use.

Concept naming. Not too much attention should be paid to the naming of the concepts. The main reason for the unusual names was an attempt not to be limited in the investigation by simply importing known concepts. We tried to formulate them by need. In this way rely-guarantee conditions appeared “naturally”. They were not part of the original method.

1 Introduction

In [10] we have discussed the strengths and weaknesses of use of Event-B for program verification. This article presents results on methodological work on developing a verification method for programs that contain both sequential and concurrent parts. Originally this work started with Event-B and basic approaches to structuring such as [9] or UML-B [11]. The intention is to preserve positive aspects of the Event-B method for proving correctness. It turns out that a promising direction is combine Event-B style verification with rely-guarantee [12, 14] thinking. For the purpose of this article we will distinguish to approaches to verification: Hoare-style *specification* (e.g. [17]) uses a proof system directly applied to the program and *refinement* [16] uses a series of abstractions of the program.

We can characterise the concept of refinement by describing what kind of contributions it can make to a program development method. In this sense refinement provides a technique for

(i) verifying program correctness,
(ii) keeping the proof effort as low as possible,
(iii) abstracting from implementation details,
(iv) preserving correctness from an abstract model to concrete models,
(v) explaining complex models,
(vi) structuring a complex proof.

Any particular development method emphasises some techniques more than others. A very strong form of (iv) is the approach of *correctness preserving transformations* where a consistent abstract models are refinement by consistent concrete models. Requiring certain consistency properties in abstract models may counteract economical proof (ii). For instance, proving that an abstract model is implementable replicates the refinement proof that provides an implementation. Nonetheless, refinement is well-suited for keeping proof effort low because properties of abstract models remain valid in concrete models. However, it matters what kinds of properties are proved in abstract models for realising this. Such considerations do not have any effect on the correctness or completeness of the method but on its practicality. Overemphasising (iii) may counteract (i) by admitting only such programs that permit the suitable abstractions. Our emphasis is on (i) and (ii). We admit formal models that lack properties that can be established by refinement. Implementability [13], also called feasibility in B [1] and Event-B [2] is such a property, for instance. Failure to find an implementation that has all required properties indicates that an abstract model may not have them either. We take abstraction to mean *less detailed* and *more nondeterministic*. We also think that a model should explain the object being modelled. This means that we require an informal interpretation of formal models that permits casting explanations in simple terms. In order to be suitable for structuring complex proofs, flexibility with respect to the choice of refinement steps is needed.

2 Similarities of specification and refinement

In specification correctness is verified by means of proof systems based on the structure of a modelling notation. Proof outlines are such a notation that comes with dedicated proof systems for program correctness. In refinement correctness is verified by showing that a series of abstractions can be brought into the form of the program to be verified. The proof processes associated with specification and refinement are very similar. Only the direction of the proofs is inverted. Specification proofs are carried out “bottom-up” whereas refinement proofs are carried out “top-down”. We illustrate by means of the program $r_1 \,; r_2$ and its proof outline

$$\{\, \varphi_0 \,\}\, r_1 \,; \{\, \varphi_1 \,\}\, r_2 \,\{\, \varphi_2 \,\}.$$

Fig. 1 shows a specification proof verifying that the program $r_1 \,;r_2$ is correct with respect to precondition φ_0 and postcondition φ_2. The proof begins verifying the subprograms r_1 and r_2 and concludes the correctness of the composed program $r_1 \,; r_2$.

$$\begin{tikzcd}
\{\, \varphi_0 \,\}\, r_1 \,\{\, \varphi_1 \,\} \arrow[dr, dash] & & \{\, \varphi_1 \,\}\, r_2 \,\{\, \varphi_2 \,\} \arrow[dl, dash] \\
& \{\, \varphi_0 \,\}\, r_1 \,; \{\, \varphi_1 \,\}\, r_2 \,\{\, \varphi_2 \,\} &
\end{tikzcd}$$

Figure 1: A specification proof for verifying the correctness the program $r_1 \,; r_2$

Fig. 2 shows corresponding the refinement proof. In order to use refinement, first an abstraction R of the program $r_1 \,; r_2$ with respect to precondition φ_0 and postcondition φ_2 has to be found. In the next step it is proved that r_0 can be obtained by combining r_1 and r_2. In a refinement proof the formula

$$\begin{tikzcd}
& \dfrac{\{\, \varphi_0 \,\}\, r_0 \,\{\, \varphi_2 \,\}}{\{\, \varphi_0 \,\}\, r_1 \,; \{\, \varphi_1 \,\}\, r_2 \,\{\, \varphi_2 \,\}} \arrow[dl, dash] \arrow[dr, dash] & \\
\{\, \varphi_0 \,\}\, r_1 \,\{\, \varphi_1 \,\} & & \{\, \varphi_1 \,\}\, r_2 \,\{\, \varphi_2 \,\}
\end{tikzcd}$$

Figure 2: A refinement proof for the correctness the program $r_1 \,; r_2$

$\{\, \varphi_0 \,\}\, r_1 \,; \{\, \varphi_1 \,\}\, r_2 \,\{\, \varphi_2 \,\}$ is specified to refine the formula $\{\, \varphi_0 \,\}\, r_0 \,\{\, \varphi_2 \,\}$ which is proved in terms of r_1 and r_2 similarly to the specification proof.

Although the similarity between specification and refinement is well known it is not well exploited in practical verification methods. B [1] enforces refinement of operation bodies as a whole where the bodies are specifications. The known problem with this approach is that proof obligations used to verify refinements are very difficult to master. In Event-B [2] this problem has been addressed by removing specifications from the method. VDM [15] supports refinement and operation decomposition, a specification technique for operation bodies based on Hoare rules similar to proof outlines. The ASM refinement [6, 7, 18] method allows also arbitrary programs as abstractions. This gives a very expressive notion of refinement but the proofs easily get difficult just as in operation refinement in B. The proof method associated with ASM is defined in an abstract algebraic fashion. It emphasis what has to be proved in terms of execution traces but does not specify how.

In the following sections we discuss a practical verification approach that combines specification and refinement in method for verifying programs consisting of sequential and concurrent behaviour. The approach is pragmatic in that specification and refinement used whenever it appears to keep formal proofs easier. The approach is more flexible than that of B or VDM but not as expressive as that of ASM. We gain some expressiveness over B and VDM by adopting concepts of rely-guarantee thinking [14] and of anticipation [3]. As opposed to ASM we do not expect refinement to reflect design decisions but only consider them as parts of complex proofs.

3 Elements of observation

As can be seen particularly well in the definition of ASM refinement it is crucial to determine what can be observed and when. We allow observation of programs from two distinct *perspectives*. From the *sequential* perspective any transition a program makes is observable. From the *concurrent* perspective only *atomic* transitions are observable. All other *compound* transition are not concurrently observable. Concurrently they appear as *stuttering*. States are generally observable within a certain *tolerance*. Tolerance makes observations potentially imprecise. A further source of imprecision is *interference*. The cause of tolerance is the program itself whereas interference may be caused by any program that could

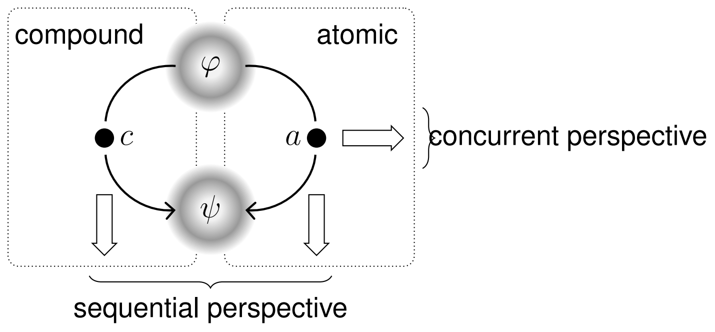

Figure 3: Observation perspectives of a compound transition c and an atomic transition a

be composed in parallel. Both are described by transitions. Interference is atomic because only atomic transitions of parallel programs are observable. Compound transitions are associated with *envelopes* that delimit the atomic transitions from which they may be composed. That is, instead of the compound transition sequences of atomic transitions taken from the envelope are observable. An envelope delimits the possible implementations of a compound transitions. Transitions are described by means of *commands*

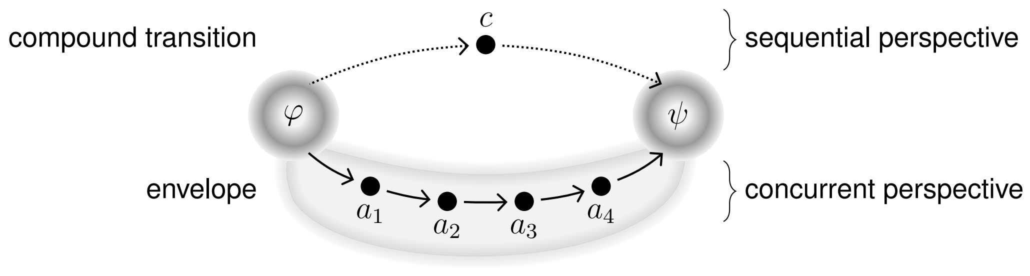

Figure 4: Concurrent observation of the envelope of a compound transition c

and states by means of *places*. They are discussed in the following Sections 3.1 and 3.2. More complex behaviour is expressed by means of *machines*. Machines are composed of places and commands. Machines are defined in terms of *sequential machines* and *parallel machines*. The two kind of machines are discussed in Sections 6 and 7, respectively.

3.1 Commands

Simple commands r are composed of a *guard* and an *action* and have the following shape,

$\mathbf{any}\ pr$	*(guard parameters)*
$\mathbf{when}\ gr$	*(guard condition)*
$\mathbf{some}\ qr$	*(action parameters)*
$\mathbf{where}\ hr$	*(action condition)*
$\mathbf{do}\ vr := er$	*(action statement)*

The *guard condition* gr constrains the states in which a command may occur. The *action condition* hr constrains the successor states determined by an *action statement* $vr := er$ where vr are variables. We often need to refer to the conjunction of guard and action condition; we let $cr \equiv gr \wedge hr$. Nondeterministic guards can be specified by means of *guard parameters* pr and nondeterministic actions be means of *action parameters* qr which must be distinct. Furthermore, parameters must be distinct from variables. Variables occurring free in a command are restricted by the *alphabet* of that command. An *alphabet* is a set of variables with associated types. Compound commands are decorated with a preceding "**:**" to distinguish them from atomic commands that are decorated with a preceding "**.**". By $\mathbf{skip}$ we denote the command "$\mathbf{when}\ true$" that is always enabled and does not change the state. Other common shapes are naked guards "$\mathbf{when}\ gr$" or naked actions "$\mathbf{do}\ vr := er$". Undecorated simple commands can be either compound or atomic. Simple command r is named rn using the notation $r \triangleq rn$. A *complex command* is a pair $R \bullet r$ where R is a set of simple commands and r is a simple command.

3.2 Places

A *place* $@\,a$ is a named assertion a. *Assertions* are first-order predicates. The tolerance T is a set of commands. Similarly to commands, assertions have alphabets. Because all variables are typed we can take the customary view that assertions describe sets of states. A *state* assigns a value to each variable of an alphabet A. By $\mathrm{free}(a)$ we denote the set of free variables occurring in assertion a. Place $@\,a$ is named φ using the notation $\varphi \triangleq @\,a$. Places φ occur in machines where they are referenced by $\{\,\varphi\,\}$. In logical formulas involving places we use $@\varphi$ to refer to assertion a.

3.3 Machines

A *basic machine* is of one of the following shapes[1]:

(i)	$P\ R \bullet r\ Q$	*(operation)*
(ii)	$P\ R \bullet r\ M$	*(sequential composition)*
(iii)	$P\ ([]_{i=1}^{m}\ R_i \bullet r_i\ M_i)$	*(choice)*
(iv)	$P\ ([]_{i=1}^{m}\ R_i \bullet r_i\ P) * ([]_{i=1}^{n}\ S_i \bullet s_i\ N_i)$	*(iteration)*
(v)	$\|_{i=1}^{n}\ O_i$	*(parallel composition)*

where P and Q are places, r, r_1, …, r_m, s_1, …, s_n are commands, R, R_1, …, R_m, S_1, …, S_n are sets of commands, M, M_1, …, M_m, N_1, …, N_n are basic machines or places, and O_1, …, O_n are *basic operations*. Basic machines are used in step-wise machine refinement to introduce the corresponding control structures. Basic machines of the shapes (i) to (iv) are called *sequential machines* and basic machines of the shape (v) are called *parallel machines*. All places of a sequential machine have the same alphabet. *Complex machines* are constructed from basic machines by replacing (some) commands by machines or complex machines. They are the result of machine refinement. A machine, basic or complex, is called *simple* if all its commands are simple.

Machines are composed from operations in the following way. In the following let X, X_1, …, X_m, Y_1, …, Y_n be commands or machines. All operations $P\ X\ Q$ are machines. If $P\ X\ Q$ is an operation and M a machine, then $P\ X\ M$ is a machine. If $P\ X_1\ M_1$, …, $P\ X_m\ M_m$ are machines, then $P\ ([]_{i=1}^{m}\ X_i\ M_i)$ is a machine. If $P\ X_1\ P$, …, $P\ X_m\ P$ are operations and If $P\ Y_1\ N_1$, …, $P\ Y_n\ N_n$ are machines, then $P\ ([]_{i=1}^{m}\ X_i\ P) * ([]_{i=1}^{n}\ Y_i\ N_i)$ is a machine. Finally, if O_1, …, O_n are operations, then $\|_{i=1}^{n}\ O_i$ is a machine. We define the *(basic) operations of a machine* M to be those (basic) operations from which it is composed.

We define initial and final places of machines inductively on machines. The initial place of $P\ X\ Q$, $P\ X\ M$, $P\ ([]_{i=1}^{m}\ X_i\ M_i)$ and $P\ ([]_{i=1}^{m}\ X_i\ P) * ([]_{i=1}^{n}\ Y_i\ N_i)$ is P. The initial places of $\|_{i=1}^{n}\ O_i$ are the initial places of the operations O_i. The final place of $P\ X\ Q$ is Q. The final places of $P\ X\ M$ are the final places of M. The final places of $P\ ([]_{i=1}^{m}\ X_i\ M_i)$ are the final places of the machines M_i. The final places of $P\ ([]_{i=1}^{m}\ X_i\ P) * ([]_{i=1}^{n}\ Y_i\ N_i)$ are the final places of the machines N_i. The final places of $\|_{i=1}^{n}\ O_i$ are the final places of the operations O_i.

Remark. The first component $[]_{i=1}^{m}\ X_i\ P$ of an iteration (iv) repeats place P, the *loop invariant*. Usually, we write iterations in the form of a choice $P\ ([]_{i=1}^{m+n}\ Z_i\ L_i)$ where for $i \in 1, \ldots, m$ we have $L_i \equiv X_i *$ and for $i \in m{+}1, \ldots, m{+}n$ we have $L_{i-m} \equiv Y_i\ N_i$. This avoids repetition of the loop invariant P.

Remark. The names given to commands and assertions in a machine M are significant. They are used to identify “locations” in the machine M. Commands r occurring in operations $P\ r\ Q$ of M and places of M are only identifiable by their names. Names of commands belonging to the tolerance of some place of M only need to be identifiable within the tolerance. We do not require that all entities occurring in a machine be identifiable but only those that need identification.

3.4 Protocols

Each machine M is associated with a *protocol* $\S\,a \bullet T \circ I$ where a is an assertion, T is the tolerance of the protocol and I its interference. Tolerance and interference are sets of simple commands. Protocols

[1] We use $[]_{i=1}^{m}\ K_i$ as an abbreviation of $K_1\ []\ \ldots\ []\ K_m$ and $\|_{i=1}^{n}\ L_i$ as an abbreviation of $L_1 \parallel \ldots \parallel L_n$.

$\S\, a \bullet T \circ I$ are named by writing $\varphi \triangleq \S\, a \bullet T \circ I$. Protocols of the shape $\S\, a \bullet T$ with empty interference are also called envelopes. When dealing with logical formulas we write $@\varphi$ to refer to the assertion of protocol φ.

Remark. In the rely-guarantee approach [14] to verifying concurrent programs tolerance would be called a guarantee condition and interference a rely condition. In our approach tolerance is associated with commands (and indirectly places) and interference with machines. We also have a more dynamic understanding of the two concepts so that using the term "condition" appears less appropriate.

4 Verification

Most of what we want to say about programs can be stated in terms of commands and assertions. We do not use special notation for variables modelling the pre or post state similar to B, Event-B or Z. The syntax of commands that we use permits us to state properties of commands by rewriting assertions. However, in our approach maintaining alphabets during refinement is essential because variables are concurrently observable and must not change. We deal with this by attaching a dash to variables repeated in concrete machines; if x is the variable occurring in the abstract machine, then x' is the variable occurring in the concrete machine. Repeated variables are related by equality $x = x'$. For an alphabet A we denote by Π_A the conjunction of the equalities $x = x'$ for all x in the alphabet A. In the term β'_A each variable x in β is replaced by x' if x is in the alphabet A and unchanged otherwise.

The other two concepts we need are variants and witnesses. They are introduced in the following paragraphs describing properties of commands. In the following paragraphs let vr be the alphabet of command r and vs be the alphabet of command s.

4.1 Consistency

The simple operation $\{\, \varphi \,\}\ r\ \{\, \psi \,\}$ is called *consistent* if it satisfies ***cns***$(@\varphi, r, @\psi)$,

$$\textbf{\textit{cns}}(a, r, b) \quad \equiv \quad a \wedge cr \Rightarrow b[vr := er].$$

That is, if assertion φ holds when command r occurs, then assertion ψ holds after r has occurred. A complex operation $\{\, \varphi \,\}\ R \bullet r\ \{\, \psi \,\}$ is called *consistent* if $\{\, \varphi \,\}\ r\ \{\, \psi \,\}$ is consistent and for all $s \in R$ also ***cns***$(@\varphi, s, @\varphi)$ holds.

Machine consistency. A machine M is called *consistent* if (1) all its basic operations $\{\, \varphi \,\}\ R \bullet r\ \{\, \psi \,\}$ are consistent and (2) for all its complex operations $\{\, \varphi \,\}\ X\ \{\, \psi \,\}$, for all initial places $\{\, \vartheta \,\}$ of X the implication $@\varphi \Rightarrow @\vartheta$ holds; and if X is a parallel machine and $\{\, \vartheta_1 \,\}, \ldots, \{\, \vartheta_n \,\}$ are the final places of X, then $@\vartheta_1 \wedge \ldots \wedge @\vartheta_n \Rightarrow @\psi$, otherwise for each final place $\{\, \vartheta \,\}$ of X the implication $@\vartheta \Rightarrow @\psi$ holds.

Protocol consistency. The protocol $\S\, \varphi \bullet T \circ I$ is called *consistent* if for each $s \in T \cup I$, the simple operation, $\{\, \varphi \,\}\ s\ \{\, \varphi \,\}$ is consistent.

4.2 Feasibility

Atomic commands are not decomposed into smaller entities. Their feasibility is not established by implementing them. Hence, feasibility needs to be established by other means. The simple operation $\{\, \varphi \,\}\ r\ \{\, \psi \,\}$ is called *feasible* if it satisfies ***fis***$(@\varphi, {\centerdot}r)$,

$$\textbf{\textit{fis}}(a, {\centerdot}r) \quad \equiv \quad a \wedge gr \Rightarrow \exists\, qr :: hr.$$

Given φ and the guard hold, the action condition must be satisfiable for some choice of action parameters. Feasibility of atomic commands is proved late in a model with the assertions φ as strong as possible. Feasibility of compound commands does not need to be proved. It is established by refinement.

4.3 Simulation

Refinement is expressed in terms of *simulation*: abstract commands s simulate concrete commands r,

$$\textbf{\textit{sim}}(a, r, s, b) \quad \equiv \quad a \wedge \mathit{II}_D \wedge cr \wedge ur = er \Rightarrow \exists\, us, ps, qs :: cs \wedge us = es \wedge (b \wedge \mathit{II}_D)[vr'_D, vs := ur, us]$$

where $D = \alpha r \cap \alpha s$ is the shared alphabet of the abstract and the concrete commands. We define: the simple operation $\{\,\eta\,\}\ r\ \{\,\vartheta\,\}$ *refines* the simple operation $\{\,\varphi\,\}\ s\ \{\,\psi\,\}$ if $@\eta = a_0 \wedge a$ with $@\varphi \Rightarrow a_0$ and $@\vartheta = b_0 \wedge b$ with $@\psi \Rightarrow b_0$ for some assertions a and b and ***sim***$(@\eta, r, s, b)$. The two conditions $@\varphi \Rightarrow a_0$ and $@\psi \Rightarrow b_0$ permit weakening assertions in refinements, thus, "forgetting" some properties that have been established in abstractions. Note that "forgetting" is restricted by the conditions of machine consistency. The definition of simulation is standard and appears, for instance, in similar form in the specification of the Event-B proof obligations [8]. The two commands r and s must either both be atomic or both be compound. In practice, simulation ***sim*** is difficult to handle. The concept of *witness* introduced in Event-B is a great improvement in this respect [2, 10]: using a witness W, simulation can be decomposed into the two formulas ***sim***$_W$ and ***con***$_W$, called *witness simulation*,

$$\textbf{\textit{sim}}_W(a, r, s, b) \quad \equiv \quad a \wedge \mathit{II}_D \wedge cr \wedge W \Rightarrow cs \wedge (b \wedge \mathit{II}_D)[vr'_D, vs := er, es]$$

and *witness connection*

$$\textbf{\textit{con}}_W(a, r, s) \quad \equiv \quad a \wedge cr \Rightarrow \exists\, ps, qs :: W.$$

The references to the abstract and concrete post states ur and us have been eliminated by applying twice the one-point rule. Often the assertion a can be formulated so that witness connection is easily proved. The conclusion of witness simulation usually is a larger conjunction that can be proved conjunct by conjunct. The conclusion of witness connection existentially quantifies over the abstract parameters ps and qs, that is, abstract and concrete commands are related by connecting their parameters; and the witness W describes the connection.

Envelope refinement. The place $\S\,\psi\ \bullet\ R\ \circ$ *refines* the place $\S\,\varphi\ \bullet\ S\ \circ$ if for each $r \in R$ there is a $s \in S \cup \{\mathbf{skip}\}$ such that $\{\,\psi\,\}\ r\ \{\,\psi\,\}$ *refines* $\{\,\varphi\,\}\ s\ \{\,\varphi\,\}$.

Operation refinement. The basic operation $\{\,\eta\,\}\ R\bullet r\ \{\,\vartheta\,\}$ *refines* the basic operation $\{\,\varphi\,\}\ S\bullet s\ \{\,\psi\,\}$ if (1) $\{\,\eta\,\}\ r\ \{\,\vartheta\,\}$ refines $\{\,\varphi\,\}\ r\ \{\,\psi\,\}$ and (2) for each $t \in R$ there is a $u \in S \cup \{\mathbf{skip}\}$ such that $\{\,\psi\,\}\ t\ \{\,\vartheta\,\}$ *refines* $\{\,\varphi\,\}\ u\ \{\,\varphi\,\}$.

Protocol adherence. Machine M *adheres* to protocol $\S\,\varphi\ \bullet\ T\ \circ\ I$ if (1) for each place $\{\,\varphi\,\}$ of M we have $\psi \Rightarrow \varphi$ and $\{\,\varphi\,\}\ s\ \{\,\varphi\,\}$ is consistent for each $s \in I$, (2) each operation $\{\,\psi\,\}\ R\bullet r\ \{\,\vartheta\,\}$ of M refines $\{\,\varphi\,\}\ T \cup \{\mathbf{.\!:keep}(\varphi)\} \bullet t\ \{\,\varphi\,\}$ for some $t \in T \cup \{\mathbf{.\!:keep}(\varphi)\}$.

Remark. Machines occurring in parallel compositions within complex machines may adhere to stricter protocols that enlarge the interference. By contrast, interference must not be reduced. For each place P of M the interference is denoted by $\mathrm{int}(P)$. The protocol of the machine determines the value of $\mathrm{int}(P)$ for all its places.

Protocol refinement. The protocol $\S\,\psi\ \bullet\ T\ \circ\ I \cup J$ refines the protocol $\S\,\varphi\ \bullet\ S\ \circ\ I$ if the envelope $\S\,\psi\ \bullet\ T \cup J$ refines the envelope $\S\,\varphi\ \bullet\ S$.

4.4 Superposition

When protocols are composed in parallel they can be developed separately provided their respective effects are taken into account. The tolerance of one protocol must match the interference of all other protocols composed in parallel. This is achieved by *superposing* their (atomic) tolerance and interference.

Superposition is a special form of refinement of where the shared alphabet is postulated to be the entire abstract alphabet and the involved assertions are invariants of the concrete and abstract commands, respectively: the simple operation $\{\,\varphi\,\}\ \textbf{.}r\ \{\,\varphi\,\}$ is superposed by the simple operation $\{\,\psi\,\}\ \textbf{.}s\ \{\,\psi\,\}$ if ***sim***$(@\varphi \wedge @\psi, \textbf{.}r, \textbf{.}s, \mathit{true})$ with $D = \alpha\textbf{.}s$, denoted by ***sup***$(@\varphi, \textbf{.}r, @\psi, \textbf{.}s)$. Using a witness W this can be decomposed into witness simulation and witness connection, where we restate witness simulation in the form of ***sup***$_W(@\varphi, \textbf{.}r, @\psi, \textbf{.}s)$,

$$\textbf{\textit{sup}}_W(a, \textbf{.}r, b, \textbf{.}s) \quad \equiv \quad a \wedge b \wedge \mathit{\Pi}_A \wedge cr \wedge W \Rightarrow cs \wedge \mathit{\Pi}_A[vr'_A, vs := er, es]$$

with $A = \alpha\textbf{.}s$.

Protocol superposition. The protocols $Z_1, \ldots, Z_n$ with $Z_k \equiv \S\,\varphi_k \bullet T_k \circ I_k$ are *superposable* if for all pairs $i \neq j$ for each $\textbf{.}t_i$ in T_i there is a $\textbf{.}t_j$ in I_j such that, considered as simple operations, $\{\,\varphi_i\,\}\ \textbf{.}t_i\ \{\,\varphi_i\,\}$ is superposed by $\{\,\varphi_j\,\}\ \textbf{.}t_j\ \{\,\varphi_j\,\}$ (with $D = \alpha\textbf{.}t_j$).

5 In the beginning

A formal model begins with the statement of a consistent operation $\{\,\varphi\,\}R\ \textbf{.:}r\ \{\,\psi\,\}S$ and an associated consistent protocol $\S\,\vartheta \bullet T \circ I$. We discuss verification of sequential and concurrent programs by means of two simple examples. The only purpose of the examples is to explain the verification method. The first example demonstrates the approach to verifying nested loops and the second the use of auxiliary variables in verifying concurrent programs.

Example NL1. We specify the simple machine "fact" for computing the factorial function by

$$\mathrm{fact} \quad \triangleq \quad \{\,\mathrm{pr}\,\}\ \textbf{.:}\mathrm{fac}\ \{\,\mathrm{pr}\,\}$$

with protocol $\S\,\mathrm{pr} \bullet \textbf{.}\mathrm{tol}$ where

$$\begin{aligned} \textbf{.:}\mathrm{fac} \quad &\triangleq \quad \mathbf{do}\ v := v! \\ \mathrm{pr} \quad &\triangleq \quad v \in \mathbb{N} \\ \textbf{.}\mathrm{tol} \quad &\triangleq \quad \mathbf{some}\ w\ \mathbf{where}\ w \in \mathbb{N}\ \mathbf{do}\ v := w \end{aligned}$$

The factorial $v!$ is to be computed in-place. The tolerance of the protocol specifies that v may be changed arbitrarily during the computation. The only interference permitted is $\textbf{.skip}$. This is a verification of an ordinary sequential program. Thus, we expect that the verification is not complicated by the presence of the concurrency constructs. Consistency of the protocol and the machine as well as protocol adherence are easily verified. □

Example AV1. The simple machine "mfoo" computes nondeterministically a value from the set $\{0, 1, 2\}$.

$$\mathrm{mfoo} \quad \triangleq \quad \{\,\mathrm{pp}\,\}\ \textbf{.:}\mathrm{foo}\ \{\,\mathrm{pp}\,\};$$

with protocol $\mathrm{pt} \triangleq \S\,v \in \mathbb{N} \bullet \textbf{.}\mathrm{tol}$ where $\textbf{.}\mathrm{tol} \triangleq \mathbf{some}\ w\ \mathbf{where}\ w \in \mathbb{N}\ \mathbf{do}\ v := w$ and

$$\begin{aligned} \textbf{.:}\mathrm{foo} \quad &\triangleq \quad \mathbf{some}\ w\ \mathbf{where}\ w \in \{0, 1, 2\}\ \mathbf{do}\ v := w \\ \mathrm{pp} \quad &\triangleq \quad @\,v \in \mathbb{N} \end{aligned}$$

The tolerance of the protocol permits arbitrary assignment of a natural number to v. Similarly to example NL1 only $\textbf{.skip}$ is permitted as interference. This is a verification of a parallel program. Nontrivial interference occurs in its component programs to be specified later on. Consistency of the protocol and the machine as well as protocol adherence are easily verified. □

6 Sequentiality

6.1 Separation

In general, compound commands can only refine compound commands and atomic commands can only refine atomic commands. Separation permits to eliminate compound commands replacing it by two atomic commands and an intermediate place. Once a compound command has been separated, refinement continues in terms of these atomic commands and the intermediate place. The machine implementing the compound command is developed from the intermediate place. In the course of the implementation the two atomic commands must be refined to $\mathbf{.skip}$. Separation is a purely syntactic transformation of an operation $O \equiv P \; \mathbf{.:}r \; Q$. Let $\S \, a \bullet T \circ I$ the protocol of $P \; \mathbf{.:}r \; Q$. Using fresh variables w, separation replaces $\mathbf{.:}r$ by

$$P \; \mathbf{.}s \; \{ \, a \, \} \; T \cup \{ \mathbf{.:keep}(a) \} \bullet \mathbf{.}t \; Q$$

where $\mathbf{.}s \triangleq w := v$ and $\mathbf{.}t \triangleq \mathbf{any} \; pr \; \mathbf{when} \; gr[v := w] \; \mathbf{some} \; qr \; \mathbf{where} \; hr[v := w] \; \mathbf{do} \; vr := er[v := w]$ are two atomic commands that must eventually be refined to $\mathbf{.skip}$. The compound command $\mathbf{.:keep}(a)$ that is included in the tolerance of the intermediate place is defined by $\mathbf{.:keep}(a) \triangleq \mathbf{some} \; u \; \mathbf{where} \; a[v := u] \; \mathbf{do} \; v := u$. We denote the separated command by $w/\mathbf{.:}r$.

Remark. We permit that variables not occurring free in gr, hr or er not to be duplicated in the list w so that in the extreme case w may be empty. In this case $\mathbf{.:}r$ is replaced by $P \; T \cup \{ \mathbf{.:keep}(a) \} \bullet \mathbf{.}t \; Q$.

Note. Convergence of compound events has to be shown before they are separated.

Remark. Separation is similar to the sequential composition rule in [16, Law 8.4]

$$w, x : [pre, post] \; \sqsubseteq \; \mathrm{con} \; X \bullet \; x : [pre, mid] \; ; \; w, x : [mid[x_0 \setminus X], post[x_0 \setminus X]]$$

where mid is a fresh predicate that describes the intermediate behaviour of the program $w, x : [pre, post]$. Note, in particular, the use of the fresh *logical constants* X. Separation introduces fresh *variables* X that are assigned in the first command and used in the second command. They cannot be modified by tolerance, thus, remain constant. Note also, that the machine resulting from separation is similar to the initial machines used in [2] for sequential program development.

Example NL2. We separate "fact", introducing a fresh variable m. Let

$$m/\mathbf{.:}\mathrm{fac} \quad \equiv \quad \{ \, \mathrm{pr} \, \} \; \mathbf{.}\mathrm{sf} \; \{ \, \mathrm{qr} \, \} \; \mathbf{.}\mathrm{tol}, \mathbf{.:keep}(@\mathrm{pr}) \bullet \mathbf{.}\mathrm{ef} \; \{ \, \mathrm{pr} \, \}$$

where $\mathbf{.}\mathrm{sf} \triangleq \mathbf{do} \; m := v$ and $\mathbf{.}\mathrm{ef} \triangleq \mathbf{do} \; v := m!$. The two atomic commands $\mathbf{.}\mathrm{sf}$ and $\mathbf{.}\mathrm{ef}$ must now be refined to $\mathbf{.skip}$. Thus the computation proper takes place in the new intermediate place $\{ \, \mathrm{pr} \, \}$. □

6.2 Composition

Complex machines are composed from basic and complex machines by refinement replacing complex commands $S \bullet r$ by basic sequential machines M. Let $P \; S \bullet r \; Q$ basic operation of a machine M. This basic operation can be refined by a basic sequential machine N, effectively replacing $S \bullet r$ in M by N. The basic operations O of N satisfy either of the following conditions:

(i) O refines $P \; s \; P$ for some $s \in S$, or

(ii) O refines $P \; r \; Q$.

Machine N is composed respecting the order of operations induced by $P \; S \bullet r \; Q$: each path of N has at most one operation that refines $P \; r \; Q$, each path of N not ending in $*$ has an operation that refines $P \; r \; Q$. each path of N ending in $*$ does not contain an operation that refines $P \; r \; Q$. The *paths* of a basic sequential machine N are defined inductively as follows:

$$\begin{aligned}
\mathrm{path}(P \; r \; Q) &= \{P \; r \; Q\}, \\
\mathrm{path}(P \; r \; M) &= \{P \; r \; K \mid K \in \mathrm{path}(M)\}, \\
\mathrm{path}(P \; ([]_{i=1}^{m} \; r_i \; M_i)) &= \textstyle\bigcup_{i=1}^{m} \{P \; r_i \; K_i \mid K_i \in \mathrm{path}(M_i)\}, \\
\mathrm{path}(P \; ([]_{i=1}^{m} \; r_i \; P) * ([]_{i=1}^{n} \; s_i \; N_i)) &= \textstyle\bigcup_{i=1}^{m} \{P \; r_i \; P \, *\} \cup \bigcup_{i=1}^{n} \{P \; r_i \; K_i \mid K_i \in \mathrm{path}(N_i)\}.
\end{aligned}$$

We write $r \sim N$ if N is to replace r. In N we indicate operation refinement by writing $s\backslash t$ where command s occurs in an operation of N and t is command from the tolerance S or it is the command r to be replaced. Command names do not need to be unique across the tolerances of all operations of M.

In order to preserve consistency, after replacing r we need to maintain the implications $\varphi \Rightarrow \psi$ between the assertions of consecutive places in the complex machine. This is achieved by replacing places with identically named refinements: if φ identifies a place, then $@\varphi \sim a$ reintroduces the place under the same name (see [11]). The new place replaces the old place of the same name everywhere in the complex machine.

Example NL3. We refine $m/\mathbf{.\!:}\mathrm{fac}$ by replacing $\mathbf{.}\mathrm{ef}$ by the machine

$$\mathbf{.}\mathrm{ef} \quad \sim \quad \{\,\mathrm{qr}\,\}\ \mathbf{.}\mathrm{tol}\backslash\mathbf{.}\mathrm{tol} \bullet \mathbf{.}\mathrm{fin}\backslash\mathbf{.}\mathrm{tol}\ \{\,\mathrm{tf}\,\}\ \mathbf{.skip}\backslash\mathbf{.}\mathrm{ef}\ \{\,\mathrm{pr}\,\}$$

where $\mathbf{.}\mathrm{fin} \triangleq \mathbf{do}\ v := m!$ and $\mathbf{.}\mathrm{tf} \triangleq \mathrm{qr} \wedge v = m!$. We continue replacing $\mathbf{.}\mathrm{fin}$ as follows

$$\mathbf{.}\mathrm{fin} \quad \sim \quad \{\,\mathrm{qr}\,\}\ \mathbf{.}\mathrm{ini}\backslash\mathbf{.}\mathrm{tol}\ \{\,\mathrm{rv}\,\}\ (\mathbf{.}\mathrm{mul}\backslash\mathbf{.}\mathrm{tol} * [\!]\ \mathbf{.}\mathrm{fin}\backslash\mathbf{.}\mathrm{fin}\ \{\,\mathrm{tf}\,\})$$

where $\mathbf{.}\mathrm{ini} \triangleq \mathbf{do}\ v, r := 1, 0$, $\mathbf{.}\mathrm{mul} \triangleq \mathbf{when}\ r < m\ \mathbf{do}\ v, r := v * (r+1), r+1$, $\mathbf{.}\mathrm{fin} \triangleq \mathbf{when}\ r = m$, $\mathrm{rv} \triangleq \mathrm{qr} \wedge 0 \le r \wedge r \le m \wedge v = r!$. We continue replacing atomic commands $\mathbf{.}\mathrm{ini}$ and $\mathbf{.}\mathrm{mul}$,

$$\begin{aligned}
\mathbf{.}\mathrm{ini} \quad &\sim \quad \{\,\mathrm{qr}\,\}\ \mathbf{.}\mathrm{ini}\backslash\mathbf{.}\mathrm{ini}\ \{\,\mathrm{rv}\,\} \\
\mathbf{.}\mathrm{mul} \quad &\sim \quad \{\,\mathrm{rv}\,\}\ \mathbf{.}\mathrm{tst}\backslash\mathbf{.skip}\ \{\,\mathrm{su}\,\}\ (\mathbf{.}\mathrm{add}\backslash\mathbf{.skip} * [\!]\ \mathbf{.}\mathrm{mul}\backslash\mathbf{.}\mathrm{mul}\ \{\,\mathrm{rv}\,\})
\end{aligned}$$

where $\mathbf{.}\mathrm{ini} \triangleq \mathbf{do}\ v, r, u, s := 1, 0, 1, 0$, $\mathbf{.}\mathrm{tst} \triangleq \mathbf{when}\ r < m$, $\mathbf{.}\mathrm{add} \triangleq \mathbf{when}\ s < r\ \mathbf{do}\ u, s := u + v, s + 1$, $\mathbf{.}\mathrm{mul} \triangleq \mathbf{when}\ s = r\ \mathbf{do}\ v, r, s := u, r+1, 0$, $\mathrm{su} \triangleq \mathrm{rv} \wedge r < m \wedge u = (s+1) * v$. Finally,

$$\begin{aligned}
\mathbf{.}\mathrm{sf} \quad &\sim \quad \{\,\mathrm{pr}\,\}\ \mathbf{.skip}\backslash\mathbf{.}\mathrm{sf}\ \{\,\mathrm{qr}\,\} \\
\mathbf{.}\mathrm{ini} \quad &\sim \quad \{\,\mathrm{qr}\,\}\ \mathbf{.}\mathrm{ini}\backslash\mathbf{.}\mathrm{ini}\ \{\,\mathrm{rv}\,\} \\
\mathbf{.}\mathrm{tst} \quad &\sim \quad \{\,\mathrm{rv}\,\}\ \mathbf{.}\mathrm{tst}\backslash\mathbf{.}\mathrm{tst}\ \{\,\mathrm{su}\,\} \\
\mathbf{.}\mathrm{fin} \quad &\sim \quad \{\,\mathrm{rv}\,\}\ \mathbf{.}\mathrm{fin}\backslash\mathbf{.}\mathrm{fin}\ \{\,\mathrm{tf}\,\}
\end{aligned}$$

where $\mathbf{.}\mathrm{ini} \triangleq \mathbf{do}\ v, i, r, u, s := 1, v, 0, 1, 0$, $\mathbf{.}\mathrm{tst} \triangleq \mathbf{when}\ r < i$, $\mathbf{.}\mathrm{fin} \triangleq \mathbf{when}\ r = i$, $@\,\mathrm{qr} \sim @\mathrm{qr} \wedge v = m$, $@\,\mathrm{rv} \sim @\mathrm{rv} \wedge m = i$. Assembling the complex machine and removing duplicate places, we have proved:

```
$\{\, v \in \mathbb{N} \,\}$ $\mathbf{skip}$ $\{\, \exists m :: v = m \,\}$
        $\mathbf{do}\ v, i, r, u, s := 1, v, 0, 1, 0\ \{\, \exists m :: v \in \mathbb{N} \wedge 0 \le r \wedge r \le m \wedge v = r! \wedge u = (s+1) * v \wedge m = i \,\}$
        $\mathbf{when}\ r < i\ \{\, \exists m :: v \in \mathbb{N} \wedge 0 \le r \wedge r \le m \wedge v = r! \wedge r < m \wedge u = (s+1) * v \,\}$
          $\mathbf{when}\ s < r\ \mathbf{do}\ u, s := u + v, s + 1$
          $*$
          $[\!]\ \mathbf{when}\ s = r\ \mathbf{do}\ v, r, s := u, r + 1, 0$
        $*$
        $[\!]\ \mathbf{when}\ r = i\ \{\, \exists m :: v \in \mathbb{N} \wedge v = m! \,\}$
        $\mathbf{skip}\ \{\, v \in \mathbb{N} \,\}$
```

refines machine “fact”. □

7 Concurrency

7.1 Composition

Parallel compositions are introduced into complex machines by replacing a compound command $\mathbf{.\!:}r$ by a parallel machine $\|_{i=1}^{n}\ O_i$ with $O_i \equiv P_i\ \mathbf{.\!:}s_i\ Q_i$. Let the compound command $\mathbf{.\!:}r$ occur within a basic simple operation $P\ S \bullet \mathbf{.\!:}r\ Q$ of a machine M with protocol $\S\,\varphi \bullet T \circ I$. The parallel machine can be introduced in sequential machine in any place where an operation refining $P\ \mathbf{.\!:}r\ Q$ could appear. The following three conditions must be satisfied:

(i) the protocol φ of P **.:**r Q must be refined by the protocol ψ_i of each O_i;
(ii) the protocols $\psi_1, \ldots, \psi_n$ are superposable;
(iii) for $i \in \{1, \ldots, n\}$ each P_i **.:**t_i Q_i must refine P **.:**r Q,

where **.:**$t_i \equiv \textbf{any}\ pr\ \textbf{when}\ \bigwedge_{j=1}^{n} gs_j\ \textbf{some}\ qr\ \textbf{where}\ \bigwedge_{j=1}^{n} hs_j\ \textbf{do}\ vs_i := es_i$

Remark. The third condition is similar to the condition used for parallel introduction in [12]. It is weaker as it permits the conjoined effect of $O_1, \ldots, O_n$ to be more deterministic. The first two conditions are similar to customary conditions of rely-guarantee proof rules. The second condition is comparable to event-model decomposition of Event-B [4].

We assume that parameters of the concrete commands are subsets of the parameters of the abstract command.

Note. The final places of $O_1, \ldots, O_n$ can be strengthened separately. The conjunction of their assertions must imply the assertion of Q to preserve machine consistency. The strengthening is not done in the composition step but in later refinements. As a consequence, introducing a parallel composition will usually focus on protocol superposition.

Example AV2. We begin by replacing **.:**foo:

$$\textbf{.:}\mathrm{foo} \quad \sim \quad \{\,\mathrm{pp}\,\}\ \textbf{.}\mathrm{set}\backslash\textbf{.skip}\ \{\,\mathrm{dn}\,\}(\{\,\mathrm{dl}\,\}\ \textbf{.:}\mathrm{fol}\backslash\textbf{.:}\mathrm{foo}\ \{\,\mathrm{pp}\,\} \parallel \{\,\mathrm{dr}\,\}\ \textbf{.:}\mathrm{for}\backslash\textbf{.:}\mathrm{foo}\ \{\,\mathrm{pp}\,\})\ \{\,\mathrm{pp}\,\}$$

where $\textbf{.}\mathrm{set} \triangleq \textbf{do}\ d := 0$, $\textbf{.:}\mathrm{fol} \triangleq \textbf{some}\ w\ \textbf{where}\ w \in \{0,1,2\}\ \textbf{do}\ v := w$, $\textbf{.:}\mathrm{for} \triangleq \textbf{some}\ w\ \textbf{where}\ w \in \mathbb{N}\ \textbf{do}\ v := w$, $@\,\mathrm{dn} \triangleq @\mathrm{pp} \wedge d = 0$, $@\,\mathrm{dl} \triangleq @\mathrm{pp}$, $@\,\mathrm{dr} \triangleq @\mathrm{dn}$. The protocol of the left component is $\mathrm{pl} \triangleq \S\, @\mathrm{pp} \wedge d \in \mathbb{Z} \bullet \textbf{.}\mathrm{tll} \circ \textbf{.}\mathrm{tlr}$ and the protocol for the right component $\mathrm{pr} \triangleq \S\, @\mathrm{pp} \wedge d \in \mathbb{Z} \bullet \textbf{.}\mathrm{tlr} \circ \textbf{.}\mathrm{tll}$, where

$$\begin{aligned} \textbf{.}\mathrm{tll} &\triangleq \textbf{do}\ v := 0 \\ \textbf{.}\mathrm{tlr} &\triangleq \textbf{where}\ d \in \{0,1\}\ \textbf{do}\ v, d := v+1, d+1 \end{aligned}$$

Now we can separate each parallel component.

$$\begin{aligned} /\textbf{.:}\mathrm{fol} &\triangleq \{\,\mathrm{dl}\,\}\ \textbf{.}\mathrm{tll}, \textbf{.:keep}(@\mathrm{pl}) \bullet \textbf{.}\mathrm{fol}\ \{\,\mathrm{pp}\,\} \\ /\textbf{.:}\mathrm{for} &\triangleq \{\,\mathrm{dr}\,\}\ \textbf{.}\mathrm{tlr}, \textbf{.:keep}(@\mathrm{pr}) \bullet \textbf{.}\mathrm{for}\ \{\,\mathrm{pp}\,\} \end{aligned}$$

We replace **.**fol:

$$\textbf{.}\mathrm{fol} \quad \sim \quad \{\,\mathrm{dl}\,\}\ \textbf{.}\mathrm{sze}\backslash\textbf{.}\mathrm{tll}\ \{\,\mathrm{ql}\,\}\ \textbf{.skip}\backslash\textbf{.}\mathrm{fol}\ \{\,\mathrm{pp}\,\}$$

where $\textbf{.}\mathrm{sze} \triangleq \textbf{do}\ v := 0$ and $@\,\mathrm{ql} \triangleq @\mathrm{dl} \wedge v \in \{0,1,2\} \wedge v \le d$. And we replace **.**for:

$$\textbf{.}\mathrm{for} \quad \sim \quad \{\,\mathrm{dr}\,\}\ \textbf{.}\mathrm{vpa}\backslash\textbf{.}\mathrm{tlr}\ \{\,\mathrm{da}\,\}\ \textbf{.}\mathrm{vpb}\backslash\textbf{.}\mathrm{tlr}\ \{\,\mathrm{db}\,\}\ \textbf{.skip}\backslash\textbf{.}\mathrm{for}\ \{\,\mathrm{pp}\,\}$$

where $\textbf{.}\mathrm{vpa} \triangleq \textbf{do}\ v, d := v+1, d+1$, $\textbf{.}\mathrm{vpa} \triangleq \mathrm{vpa}$, $@\,\mathrm{da} \triangleq @\mathrm{pp} \wedge d = 1$, $@\,\mathrm{db} \triangleq @\mathrm{pp}$. The problem that remains is to remove the auxiliary variable d. We do this by dropping the interference conditions and proving further refinements in the Owicki/Gries method. □

7.2 Exposure

In the specification method there are methods for removing auxiliary variables, e.g., [5] or for rely-guarantee, e.g, [20]. If we were to continue in a refinement setting, we would now have to introduce assumptions from there components. We would have to expose more and more of the internals of the different components. The auxiliary variable rule in [20] is based on the assumption that at some point (the moment of the composition) all components are known. We use the same assumption but for dropping the interference conditions exposing all internals of the other components. Now interference proofs have to be carried out in the Owicki/Gries method. Sticking to protocols would complicate the verification proof unnecessarily. Note, that we have already achieve what rely-guarantee was intended to achieve: Our verification will not fail late because of unconsidered interference proofs. We have profited maximally from compositionally at this point but now we do not need it anymore.

Example AV3. We drop the protocols pl and pr. We refine all commands removing d. All required proofs are trivial. □

$$\textbf{CS}\!: \frac{}{\langle\sigma, P\ L\rangle \to \langle\sigma, L\rangle} \qquad \textbf{CP}\!: \frac{}{\langle\sigma, (\|_{i=1}^{n}\ P)\ L\rangle \to \langle\sigma, L\rangle} \qquad \textbf{TT}\!: \frac{}{\langle\sigma, L\rangle \to \langle\sigma, L\rangle}$$

$$\textbf{NO}\!: \frac{\mathbf{.}s \in S \quad (\sigma,\tau) \in \llbracket \mathbf{.}s \rrbracket}{\langle\sigma, P\ S \bullet r\ Q\rangle \to \langle\tau, P\ S \bullet r\ Q\rangle} \qquad \textbf{PG}\!: \frac{(\sigma,\tau) \in \llbracket \mathbf{.}r \rrbracket}{\langle\sigma, P\ S \bullet \mathbf{.}r\ Q\rangle \to \langle\tau, Q\rangle} \qquad \textbf{SQ}\!: \frac{\langle\sigma, M\rangle \to \langle\tau, N\rangle}{\langle\sigma, M\ L\rangle \to \langle\tau, N\ L\rangle}$$

$$\textbf{CD}\!: \frac{\sigma \in \mathrm{dom}(\llbracket r_i \rrbracket)}{\langle\sigma, P\ ([\!]_{i=1}^{m}\ R_i \bullet r_i\ M_i)\rangle \to \langle\sigma, P\ r_i\ M_i\rangle} \qquad \textbf{PA}\!: \frac{\langle\sigma, M_i\rangle \to \langle\tau, N_i\rangle \quad \forall\, j : j \neq i : M_j \equiv N_j}{\langle\sigma, \|_{i=1}^{n}\ M_i\rangle \to \langle\tau, \|_{i=1}^{n}\ N_i\rangle}$$

$$\textbf{LP}\!: \frac{\sigma \in \mathrm{dom}(\bigcup_{i=1}^{m} \llbracket r_i \rrbracket)}{\langle\sigma, P\ ([\!]_{i=1}^{m}\ R_i \bullet r_i\ M_i) * ([\!]_{i=1}^{n}\ S_i \bullet s_i\ N_i)\rangle \to \langle\sigma, P\ ([\!]_{i=1}^{m}\ R_i \bullet r_i\ M_i)\ ([\!]_{i=1}^{m}\ R_i \bullet r_i\ M_i) * ([\!]_{i=1}^{n}\ S_i \bullet s_i\ N_i)\rangle}$$

$$\textbf{EX}\!: \frac{\sigma \in \mathrm{dom}(\bigcup_{i=1}^{n} \llbracket s_i \rrbracket)}{\langle\sigma, P\ ([\!]_{i=1}^{m}\ R_i \bullet r_i\ M_i) * ([\!]_{i=1}^{n}\ S_i \bullet s_i\ N_i)\rangle \to \langle\sigma, P\ ([\!]_{i=1}^{n}\ S_i \bullet s_i\ N_i)\rangle}$$

$$\textbf{RP}\!: \frac{\mathrm{env}(\mathbf{.\!:}r) = Q \quad \mathbf{.}t \equiv \mathbf{any}\ pr\ \mathbf{when}\ gr[v := \sigma]\ \mathbf{some}\ qr\ \mathbf{where}\ hr[v := \sigma] \wedge vr = e[v := \sigma]}{\langle\sigma, P\ R \bullet \mathbf{.\!:}r\ R\rangle \to \langle\sigma, Q\ \mathbf{.}t\ R\rangle}$$

$$\textbf{IN}\!: \frac{\mathbf{.}r \in \mathrm{int}(P) \quad (\sigma,\tau) \in \llbracket \mathbf{.}r \rrbracket}{\langle\sigma, P\rangle \to \langle\tau, P\rangle}$$

Figure 5: Transition rules of the operational semantics of machines

8 Operational semantics

A configuration is a pair $\langle\sigma, M\rangle$ consisting of a state σ, a predicate φ and a machine M. Let $\llbracket\varphi\rrbracket$ be the set

$$\llbracket\varphi\rrbracket = \{v \mid \exists\, z :: \varphi\}$$

where z are the free variables of $\exists\, v :: \varphi$. A configuration $\langle\sigma, M\rangle$ is called *consistent* if $\sigma \in \llbracket\vartheta\rrbracket$, where $\vartheta = \varphi$ if $M = \{\,\varphi\}T\ L$ and $\vartheta = \bigwedge_{i=1}^{n} \varphi_i$ if $M = \|_{i=1}^{n}\ \{\,\varphi_i\}T_i\ L_i$. Let $\llbracket r\rrbracket$ be the relation

$$\llbracket r\rrbracket = \{v \mapsto w \mid \exists\, \mathit{pr} :: \mathit{gr} \wedge \exists\, \mathit{qr} :: \mathit{hr} \wedge w = \mathit{er}_{+}\},$$

where er_{+} is er extended to match the alphabet $\{v\}$. A configuration of the shape $\langle\sigma, P\rangle$ is called *final*.

Verifying consistency of a machine M we establish that beginning in a consistent configuration all successor configuration are consistent. By showing feasibility of atomic commands $\mathbf{.}r$ we ensure that $\llbracket\mathbf{.}r\rrbracket$ is not empty in transition rule **PG**.

Separation establishes that the atomic command $\mathbf{.}t$ of transition rule **RP** will match transition rule **TT**: because the atomic command $\mathbf{.}t$ introduced by separation is eventually refined to $\mathbf{.}\mathbf{skip}$. This behaviour must already be possible for the atomic command $\mathbf{.}t$ of transition rule **RP**. The atomic command $\mathbf{.}s$ introduced by separation plays the role of recording the initial state σ of transition rule **RP**.

Sequential refinement of an operation $P\ R{\bullet}r\ Q$ replaces $R{\bullet}r$ by a machine X with the final places Q_1, $\ldots$, Q_n preceded all other places $P_1, \ldots, P_m$. All operations in X of the form $P_i\ t\ P_j$ refine r's tolerance R or $\mathbf{.}\mathbf{skip}$, operations of the form $P_i\ \mathbf{.}s\ Q_j$ refine r. For $\mathbf{.}s$ and atomic commands $\mathbf{.}t$ this corresponds to the behaviour of $P\ \mathbf{.}r\ Q$ described by **NO**, **PG** and **TT** in the abstraction. For compound commands $\mathbf{.\!:}t$ by transition rule **RP** this corresponds to the behaviour **NO** with respect to $P\ R \bullet r\ Q$ in the abstraction. Consistency of X is established by simulation. Whenever **CD**, **LP** or **EX** apply to a configuration of X, transition rule **TT** matches this for $P\ R \bullet r\ Q$.

Parallel refinement of an operation $P\ \mathbf{.\!:}r\ Q$ introduces n parallel compound commands that conjoinedly refine $\mathbf{.\!:}r$. The tolerance of the components are composed of refinements of the tolerance of $\mathbf{.\!:}r$. Interference is introduced that refines the tolerance and interference of $P\ \mathbf{.\!:}r\ Q$ or $\mathbf{.}\mathbf{skip}$. Interference of $P\ \mathbf{.\!:}r\ Q$ is preserved. Consistency is treated by means of superposition. Superposition corresponds to considering parallel components separately using transition rule **IN**. The components can be composed, for example, using the approach of [19] such that **PA** applies for the parallel composition. Using this approach interference of one component is matched with tolerance and operations of another. Consistency is ensured because interference is an abstraction of tolerance (and operations). Exposure is only used for parallel compositions but not for separate components. The interference is removed at the parallel composition where it was introduced and everywhere inside it. Consistency of configurations obtained from applying **PA** is established by considering tolerance (and operations) directly.

References

[1] Jean-Raymond Abrial (1996): *The B-Book: Assigning Programs to Meanings*. Cambridge University Press.

[2] Jean-Raymond Abrial (2010): *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

[3] Jean-Raymond Abrial, Dominique Cansell & Dominique Méry (2005): *Refinement and Reachability in Event B*. In Helen Treharne, Steve King, Martin C. Henson & Steve Schneider, editors: *ZB05*, *Lecture Notes in Computer Science* 3455, Springer, pp. 222–241.

[4] Jean-Raymond Abrial & Stefan Hallerstede (2007): *Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B*. *Fundam. Inform* 77(1-2), pp. 1–28. Available at `http://iospress.metapress.com/content/c74274t385t6r72r/`.

[5] K. R. Apt, F. S. de Boer & E.-R. Olderog (2009): *Verification of Sequential and Concurrent Programs, 3rd Edition*. Texts in Computer Science, Springer-Verlag. 502 pp, ISBN 978-1-84882-744-8.

[6] Egon Börger & Robert Stärk (2003): *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer.

[7] Egon Börger (2003): *The ASM Refinement Method*. *Formal Aspects of Computing* 15, pp. 237–257.

[8] Stefan Hallerstede (2005): *The Event-B Proof Obligation Generator*. Technical Report, ETH Zürich.

[9] Stefan Hallerstede (2010): *Structured Event-B Models and Proofs*. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau & Steve Reeves, editors: *ABZ*, *Lecture Notes in Computer Science* 5977, Springer, pp. 273–286. Available at `http://dx.doi.org/10.1007/978-3-642-11811-1`.

[10] Stefan Hallerstede & Michael Leuschel (2012): *Experiments in program verification using Event-B*. *Formal Asp. Comput* 24(1), pp. 97–125. Available at `http://dx.doi.org/10.1007/s00165-011-0205-4`.

[11] Stefan Hallerstede & Colin F. Snook (2011): *Refining Nodes and Edges of State Machines*. In Shengchao Qin & Zongyan Qiu, editors: *ICFEM*, *Lecture Notes in Computer Science* 6991, Springer, pp. 569–584. Available at `http://dx.doi.org/10.1007/978-3-642-24559-6`.

[12] I. J. Hayes, C. B. Jones & R. J. Colvin (2012): *Refining rely-guarantee thinkings*. Technical Report, School of Computing Science, University of Newcastle upon Tyne.

[13] Eric C. R. Hehner (1993): *A Practical Theory of Programming*. Texts and Monographs in Computer Science, Springer-Verlag.

[14] Cliff B. Jones (1983): *Tentative Steps Toward a Development Method for Interfering Programs*. *TOPLAS* 5(4), pp. 596–619.

[15] Cliff B. Jones (1990): *Systematic Software Development Using VDM*, 2nd edition. Prentice Hall.

[16] Carroll C. Morgan (1994): *Programming from Specifications: Second Edition*. Prentice Hall.

[17] John C. Reynolds (1981): *The Craft of Programming*. Prentice-Hall International.

[18] Gerhard Schellhorn (2005): *ASM refinement and generalizations of forward simulation in data refinement: a comparison*. *Theor. Comput. Sci* 336(2-3), pp. 403–435.

[19] Ketil Stølen (1991): *A Method for the Development of Totally Correct Shared-State Parallel Programs*. In J. C. M. Baeten & J. F. Groote, editors: *CONCUR '91: 2nd International Conference on Concurrency Theory*, *Lecture Notes in Computer Science* 527, Springer-Verlag, Amsterdam, The Netherlands, pp. 510–525.

[20] Qiwen Xu, Willem P. de Roever & Jifeng He (1997): *The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs*. *Formal Aspects of Computing* 9(2), pp. 149–174.