Application of the Composition Principle to Unity-like Specifications

Pierre Collette*

Université Catholique de Louvain, Unité d'Informatique, Place Sainte-Barbe, B-1348 Louvain-la-Neuve, Belgium

Abstract. The problem of composing mutually dependent rely-guarantee specifications arises in the hierarchical development of reactive or concurrent systems. The composition principle has been proposed as a logic-independent solution to this problem. In this paper, we apply it to Unity-like rely-guarantee specifications. For that purpose, we interpret Unity formulas in Abadi and Lamport's compositional model. Then, the premises of the composition rule are reduced to proof obligations that can be carried out in the existing Unity proof system. The approach is illustrated by an example, the composition of mutually dependent specifications of concurrent buffers.

1 Introduction

Several specification methods [2, 10, 14] for the development of reactive or concurrent systems may be classified as rely-guarantee or assumption-commitment methods. Intuitively, a rely-guarantee specification $R \Rightarrow G$ states that a system satisfies the guarantee condition G if it operates in an environment that satisfies a rely condition R. We consider specification triples (R, G, G^S) where the safety condition G^S is implied by the full (including liveness) guarantee condition G and R is restricted to a safety condition.

Hierarchical specification methods for concurrent systems generally require composition rules. In the rely-guarantee paradigm, the composition principle of [2, 14] provides a way of combining mutually dependent specifications. If $[\![S]\!]$ denotes the set of behaviours allowed by a specification S, this principle may be stated as follows:

$$\frac{\begin{array}{lll} P_1 \text{ sat } (R_1 \Rightarrow G_1) & [\![R]\!] \cap [\![G_2^S]\!] \subseteq [\![R_1]\!] & [\![R]\!] \cap [\![G_1^S]\!] \cap [\![G_2^S]\!] \subseteq [\![G^S]\!] \\ P_2 \text{ sat } (R_2 \Rightarrow G_2) & [\![R]\!] \cap [\![G_1^S]\!] \subseteq [\![R_2]\!] & [\![R]\!] \cap [\![G_1]\!] \cap [\![G_2]\!] \subseteq [\![G]\!] \end{array}}{P_1 || P_2 \text{ sat } (R \Rightarrow G)}$$

Basically, the premises correspond the reliance, co-existence, guarantee and strength proof obligations of [10, 20]. Informally, they read:

1. Reliance/Co-existence: $[\![R]\!] \cap [\![G_2^S]\!] \subseteq [\![R_1]\!]$. P_1 does not rely on more than $P_1 || P_2$ does, nor on more than P_2 guarantees.
2. Reliance/Co-existence: $[\![R]\!] \cap [\![G_1^S]\!] \subseteq [\![R_2]\!]$. P_2 does not rely on more than $P_1 || P_2$ does, nor on more than P_1 guarantees.

* National Fund for Scientific Research (Belgium)

3. Guarantee: $[\![R]\!] \cap [\![G_1^S]\!] \cap [\![G_2^S]\!] \subseteq [\![G^S]\!]$. Under the assumptions on the environment of $P_1 || P_2$, the safety guarantee conditions of P_1 and P_2 must imply the safety guarantee conditions of $P_1 || P_2$.
4. Strength: $[\![R]\!] \cap [\![G_1]\!] \cap [\![G_2]\!] \subseteq [\![G]\!]$. Under the assumptions on the environment of $P_1 || P_2$, the guarantee conditions of P_1 and P_2 must imply the guarantee condition of $P_1 || P_2$, especially the liveness guarantee conditions.

These four conditions are stated exclusively in semantic terms (sets of allowed behaviours). Our objective is to apply them in a particular development framework. The language we have chosen is Unity logic [6] because it yields workable specifications that may be scaled up to specify large problems [15, 16, 17]. Its operators `initially`, `unless`, and `leadsto` specify initial conditions, next-state relations, and liveness requirements respectively.

Thus, the aim of this paper is to show how the composition rule may be applied to Unity-like specifications. More precisely, we interpret the specifications in Abadi and Lamport's compositional model [2, 4] and then restate the above conditions in terms of the Unity proof obligations $R, G_2^S \vdash R_1$, $R, G_1^S \vdash R_2$, $R, G_1^S, G_2^S \vdash G^S$, and $R, G_1, G_2 \vdash G$.

As discussed in [2, 4], soundness of the composition principle is reached under the hypotheses that G^S and R respectively constrain the specified system and its environment. Therefore, to reach soundness, we propose a new version of `unless` which distinguishes system from environment transitions. Essentially, this modification is similar to what is done in [3, 5] when designing compositional versions of temporal logic.

Throughout the paper, we *preserve* the Unity style of reasoning about specifications and *reuse* the Unity proof rules. This work should thus not be viewed as 'yet another language' but rather as an attempt to combine Abadi-Lamport's work [2] and Chandy-Misra's work [6, 15].

2 Logic

In this section, we interpret Unity-like specifications in Abadi and Lamport's semantic model. Then, we recall some inference rules.

2.1 Semantic Model

In temporal-logic based approaches, the set of variables is usually divided into two classes: the class of *dynamic* variables and the class of *static* variables. Dynamic variables (also called state variables) represent quantities that can vary with time, like x in the Hoare triple $\{x = n\} x := x + 1 \{x > n\}$. A *state* is then defined as a function assigning to each dynamic variable a value in its domain. In contrast, static variables represent quantities that remain constant with time, like n in the above Hoare triple. A *static valuation* is then defined as a function assigning to each static variable a value in its domain.

Abadi and Lamport interpret a specification S as a set $[\![S]\!]$ of allowed behaviours [2]. A behaviour is a sequence

$$\sigma = s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots$$

where each s_k is a state, as defined above, and each a_k is an *agent*. By convention, $|\sigma|$, $s_k.\sigma$, $a_k.\sigma$, and $\sigma|_k$ denote the length of σ, the kth state of σ, the agent responsible for the kth transition, and the finite prefix of σ ending with $s_k.\sigma$ respectively. Agents must be thought as the entities responsible for state transitions. Although two agents (program and environment) would suffice in any particular specification, considering sets of agents eases the composition problem because the parallel composition of programs corresponds to the union of their composing agents. As discussed in [2], it may help the reader to think of the agents as elementary circuit components or individual machine-language instructions but the actual identity of the individual agents never matters. What matters is whether an agent belongs to the specified system or to its environment. Let μ be a set of agents and $\overline{\mu}$ be its complement. Informally, the correctness formula $P\ sat_\mu\ S$ is valid if all the behaviours of P are allowed by $[\![S]\!]$ when the agents in μ are considered to form the program P and the agents in $\overline{\mu}$ are considered to form its environment.

2.2 Syntax and Semantics

Since Abadi and Lamport's model includes both program agents and environment agents, the logic must include formulas which distinguish transitions performed by different sets of agents. For that purpose, we replace the operator `unless` with the operator $\texttt{unless}_\mu$ to obtain formulas that constrain the transitions performed by agents in μ only. Informally, $\sigma \in [\![p\ \texttt{unless}_\mu\ q]\!]$ if and only if every transition of σ performed by a μ-agent transforms a $p \wedge \neg q$ state into a $p \vee q$ state. In other words, if $p \wedge \neg q$ holds before a μ-transition, then p holds after the transition, unless q holds. For example, if x and n are respectively dynamic and static variables, the formula

$$x = n\ \texttt{unless}_\mu\ x > n$$

asserts that a μ-agent may not decrease the value of x.

Throughout the definitions of this section, p, q are first-order assertions, and the notation $s \models_\xi p$ indicates that p holds on state s, under the static valuation ξ. Note that ξ is always submitted to a universal quantification, meaning that a Unity specification must hold for all the possible assignments to the static variables. In the above example, n may take any arbitrary value.

$$[\![p\ \texttt{unless}_\mu\ q]\!] = \{\sigma \mid \forall\xi : \forall k < |\sigma| : (s_k.\sigma \models_\xi p \wedge \neg q) \wedge (a_k.\sigma \in \mu) \Rightarrow (s_{k+1}.\sigma \models_\xi p \vee q)\}$$

In addition to `unless` used for specifying safety properties, the Unity logic is built upon a second operator `leadsto` used for specifying liveness properties. The formula $p\ \texttt{leadsto}\ q$ is equivalent to $\Box(p \Rightarrow \Diamond q)$ in temporal logic [13]: if p holds at some point in a behaviour, then q eventually holds. For example, if x and n are respectively dynamic and static variables, the formula

$$x = n\ \texttt{leadsto}\ x = 2n$$

asserts that x eventually doubles. Finally, the operator `initially` specifies initial conditions.

$$[\![p\ \texttt{leadsto}\ q]\!] = \{\sigma \mid \forall\xi : \forall k \leq |\sigma| : (s_k.\sigma \models_\xi p) \Rightarrow (\exists j : k \leq j \leq |\sigma| : s_j.\sigma \models_\xi q)\}$$
$$[\![\texttt{initially}\ p]\!] = \{\sigma \mid \forall\xi : s_1.\sigma \models_\xi p\}$$

Although three operators suffice, it is helpful to introduce the following shorthands:

$$\texttt{stable}_\mu\, p \equiv p\, \texttt{unless}_\mu\, false$$
$$\texttt{constant}_\mu\, e \equiv \texttt{stable}_\mu\, e{=}n$$

where n is a static variable of the same sort as the expression e. Intuitively, $\texttt{stable}_\mu\, p$ asserts that no μ-agent violates a p state (p is a μ-invariant [3]), and $\texttt{constant}_\mu\, e$ asserts that no μ-agent modifies the value of an expression e.

$$[\![\texttt{stable}_\mu\, p]\!] = \{\sigma \mid \forall \xi : \forall k < |\sigma| : (s_k.\sigma \models_\xi p) \land (a_k.\sigma \in \mu) \Rightarrow (s_{k+1}.\sigma \models_\xi p)\}$$

The Unity operator `invariant` will never appear in rely-guarantee specifications; it will appear in proofs only. Intuitively, an assertion is an invariant if it holds on every state of a behaviour.

$$[\![\texttt{invariant}\, p]\!] = \{\sigma \mid \forall \xi : \forall k \le |\sigma| : s_k.\sigma \models_\xi p\}$$

The generalisation to a set $S = \{f_i\}$ of formulas is straightforward:

$$[\![S]\!] = \cap_i [\![f_i]\!]$$

2.3 Proof Rules

As mentioned in the introduction, we reuse the Unity proof system. Among others, the following weakening, transitivity, conjunction and progress-safety-progress rules are directly drawn from the corresponding rules in [6]:

$$\frac{p\, \texttt{unless}_\mu\, q \quad \texttt{invariant}\, q \Rightarrow r}{p\, \texttt{unless}_\mu\, r} \qquad \frac{p\, \texttt{leadsto}\, q, \quad q\, \texttt{leadsto}\, r}{p\, \texttt{leadsto}\, r}$$

$$\frac{p_1\, \texttt{unless}_\mu\, q_1, \quad p_2\, \texttt{unless}_\mu\, q_2}{p_1 \land p_2\, \texttt{unless}_\mu (p_1 \land q_2) \lor (p_2 \land q_1) \lor (q_1 \land q_2)}$$

$$\frac{p\, \texttt{leadsto}\, q, \quad r\, \texttt{unless}_\mu\, b, \quad r\, \texttt{unless}_{\overline{\mu}}\, b}{p \land r\, \texttt{leadsto}\, (q \land r) \lor b}$$

Similarly, the union theorem of [6] yields the union and decomposition rules:

$$\frac{p\, \texttt{unless}_{\mu_1}\, q, \quad p\, \texttt{unless}_{\mu_2}\, q}{p\, \texttt{unless}_{\mu_1 \cup \mu_2}\, q} \qquad \frac{p\, \texttt{unless}_\mu\, q}{p\, \texttt{unless}_\nu\, q}\, \nu \subseteq \mu$$

The construction rules for invariants are:

$$\frac{\models p}{\texttt{invariant}\, p} \qquad \frac{\texttt{initially}\, p, \quad \texttt{stable}_\mu\, p, \quad \texttt{stable}_{\overline{\mu}}\, p}{\texttt{invariant}\, p}$$

where $\models p$ means that p is a valid assertion.

All the rules can be proved sound w.r.t. the semantic model: $[\![f_1]\!] \cap \ldots [\![f_n]\!] \subseteq [\![g]\!]$ holds for any rule with premises $f_1, \ldots, f_n$ and conclusion g.

3 Rely-Guarantee

In this section, we first recall the meaning of a rely-guarantee specification $R \Rightarrow G$. Then, we show how the subscripted `unless` formulas of Sect. 2 can be used to restrict a safety specification to system or environment transitions. Finally, we give a rely-guarantee specification of a concurrent buffer.

3.1 Definitions

Let R, G be sets of formulas. Then, the rely-guarantee specification $R \Rightarrow G$ asserts that G holds when R holds.

$$[\![R \Rightarrow G]\!] = \{\sigma \mid \sigma \in [\![R]\!] \Rightarrow \sigma \in [\![G]\!]\}$$

To make sense, the rely condition R should restrict the environment transitions only [2]. Therefore, when specifying a system composed of μ-agents, the rely condition R is restricted to `initially` and $\texttt{unless}_{\overline{\mu}}$ formulas. Formally, $R \trianglelefteq \overline{\mu}$ holds (R does not constrain μ [3, 8]):

$$R \trianglelefteq \overline{\mu} \;\text{ iff }\; \forall \sigma : \forall k < |\sigma| : \sigma|_k \in [\![R]\!] \wedge a_k.\sigma \in \mu \Rightarrow \sigma|_{k+1} \in [\![R]\!]$$

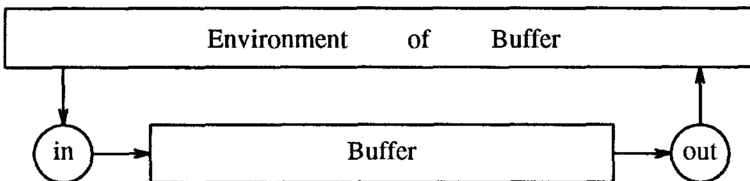
By convention, G^S denotes the safety formulas of G. Since G^S should restrict the system transitions only, we use $\texttt{unless}_\mu$ formulas in G^S when specifying a system composed of μ-agents. No `initially` formula may appear in G^S, that is initial states are determined by the environment (included in the rely condition), not by the system. Formally, $G^S \lhd \mu$ holds (G^S constrains at most μ [3, 4]):

$$G^S \trianglelefteq \mu \;\text{ iff }\; \forall \sigma : \forall k < |\sigma| : \sigma|_k \in [\![G^S]\!] \wedge a_k.\sigma \notin \mu \Rightarrow \sigma|_{k+1} \in [\![G^S]\!]$$
$$G^S \lhd \mu \;\text{ iff }\; G^S \trianglelefteq \mu \text{ and } \forall \sigma : \sigma|_1 \in [\![G^S]\!]$$

Due to the use of a unique agent symbol, such specifications distinguish between the system and its environment but not between their composing agents (μ-abstractness is preserved [2]).

3.2 Example

To highlight the use of subscripted formulas, we specify a bounded concurrent buffer that we assume to be composed of μ-agents. A buffer receives messages in the variable *in* and eventually produces them into the variable *out*.



The specification $R \Rightarrow G$ says that if the environment respects some access protocol (rely condition R), then the buffer behaves properly (guarantee condition G): no message is lost, no message is created, messages are delivered in the right order, at most N messages are buffered, input messages are eventually consumed, and buffered messages are eventually produced.

As in [1, 15, 18], we introduce an *auxiliary* variable b to represent the sequence of messages currently buffered. The formula $\mathtt{constant}_{\overline{\mu}}\ b$ asserts that b is local to the buffer, hence not modified by $\overline{\mu}$ agents. For the sake of simplicity, this formula has been stated explicitly in the examples. However, it could be considered as a part of the semantics of the auxiliary variables, hence implicitly defined. Formally, the variable b is submitted to existential quantification [1,4]; consequently, $G^S \lhd \mu$ holds although it includes the formulas $\mathtt{initially}\ b = \bot$ and $\mathtt{constant}_{\overline{\mu}}\ b$.

The static variable m ranges over messages, the special value $\bot$ denotes the absence of messages, and the operator $\bullet$ stands for sequence concatenation. Observe how the formula $\mathtt{constant}_{\mu}\ \langle in \rangle \bullet b \bullet \langle out \rangle$ suffices to specify that no message is either created or lost by the buffer and that messages are delivered in the right order[2].

Specification $\mathbf{R} \Rightarrow \mathbf{G}$

Rely R	**Guarantee G**	**Aux.Variable**
$\mathtt{initially}\ in = \bot$	$in = m\ \mathtt{unless}_{\mu}\ in = \bot$	$\mathtt{initially}\ b = \bot$
$\mathtt{initially}\ out = \bot$	$\mathtt{stable}_{\mu}\ in = \bot$	$\mathtt{constant}_{\overline{\mu}}\ b$
$out = m\ \mathtt{unless}_{\overline{\mu}}\ out = \bot$	$\mathtt{stable}_{\mu}\ out = m$	
$\mathtt{stable}_{\overline{\mu}}\ out = \bot$	$\mathtt{stable}_{\mu}\ \lvert b \rvert \leq N$	
$\mathtt{stable}_{\overline{\mu}}\ in = m$	$\mathtt{constant}_{\mu}\ \langle in \rangle \bullet b \bullet \langle out \rangle$	
	$\lvert b \rvert < N\ \mathtt{leadsto}\ in = \bot$	
	$\lvert b \rvert > 0\ \mathtt{leadsto}\ out \neq \bot$	

The Unity style is preserved in writing specifications as well as in reasoning about them. For example, the following deductions can be rewritten from [15]:

$$G^S \vdash \mathtt{invariant}\ \lvert b \rvert \leq N \tag{1}$$

$$G^S \vdash out = \bot\ \mathtt{unless}_{\mu}\ \lvert b \rvert < N \vee in = \bot \tag{2}$$

$$H, G \vdash out = \bot\ \mathtt{leadsto}\ \lvert b \rvert < N \vee in = \bot \tag{3}$$

where $H = \{\mathtt{stable}_{\overline{\mu}}\ out = \bot\}$. Detailed proof examples will be given in next section.

4 Composition

In this section, we show that the syntactic restrictions imposed on the rely-guarantee specifications of Sect. 3 match the hypotheses of the composition rule. We then put this rule into practice by replacing the premises expressed in semantic terms with suitable Unity proof obligations. We finally illustrate the rule by composing mutually-dependent rely-guarantee specifications of concurrent buffers.

[2] It also ensures that b does not introduce infinite invisible nondeterminism, a sufficient condition for existential quantification to preserve safety [1]

4.1 Parallel Composition Rule

Let $\mu = \mu_1 \cup \mu_2$ with $\mu_1 \cap \mu_2 = \emptyset$. Formulated in terms of correctness formulas, the composition principle becomes the rule

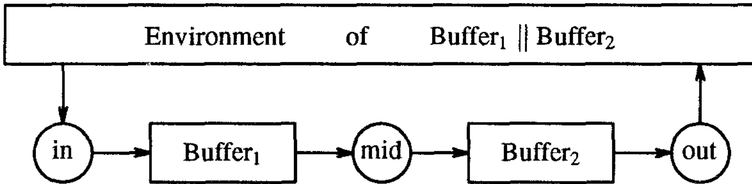
$$\frac{\begin{array}{lll} P_1 \text{ sat}_{\mu_1} (R_1 \Rightarrow G_1) & [\![R]\!] \cap [\![G_2^S]\!] \subseteq [\![R_1]\!] & [\![R]\!] \cap [\![G_1^S]\!] \cap [\![G_2^S]\!] \subseteq [\![G^S]\!] \\ P_2 \text{ sat}_{\mu_2} (R_2 \Rightarrow G_2) & [\![R]\!] \cap [\![G_1^S]\!] \subseteq [\![R_2]\!] & [\![R]\!] \cap [\![G_1]\!] \cap [\![G_2]\!] \subseteq [\![G]\!] \end{array}}{P_1 || P_2 \text{ sat}_{\mu} (R \Rightarrow G)}$$

According to [2], this rule is sound under the hypotheses $R_1 \trianglelefteq \overline{\mu_1}$, $G_1^S \lhd \mu_1$, $R_2 \trianglelefteq \overline{\mu_2}$, $G_2^S \lhd \mu_2$, $R \trianglelefteq \overline{\mu}$, and $G^S \lhd \mu$. As stated in Sect. 3, a very simple syntactic restriction suffices: the subscripts appearing in the `unless` formulas of R_1, G_1^S, R_2, G_2^S, R, and G^S are $\overline{\mu_1}$, μ_1, $\overline{\mu_2}$, μ_2, $\overline{\mu}$, and μ respectively. Since the Unity proof system is sound, we may replace set inclusion with proof obligations in the premises:

$$\frac{\begin{array}{lll} P_1 \text{ sat}_{\mu_1} (R_1 \Rightarrow G_1) & R, G_2^S \vdash R_1 & R, G_1^S, G_2^S \vdash G^S \\ P_2 \text{ sat}_{\mu_2} (R_2 \Rightarrow G_2) & R, G_1^S \vdash R_2 & R, G_1, G_2 \vdash G \end{array}}{P_1 || P_2 \text{ sat}_{\mu} (R \Rightarrow G)}$$

4.2 Specification of Concurrent Buffers

$Buffer_1$ transmits messages from the variable in to the intermediate variable mid and $Buffer_2$ transmits messages from the variable mid to the variable out. The specifications $R_1 \Rightarrow G_1$ and $R_2 \Rightarrow G_2$ are drawn from the specification $R \Rightarrow G$ by suitable renaming. Since $\mu_2 \subseteq \overline{\mu_1}$ and $\mu_1 \subseteq \overline{\mu_2}$, the rely conditions R_1, R_2 depend on the guarantee conditions G_2, G_1 respectively. So the proposed rely-guarantee specifications are mutually dependent.



Specification $\mathbf{R_1} \Rightarrow \mathbf{G_1}$

Rely $\mathbf{R_1}$	**Guarantee** $\mathbf{G_1}$	**Aux.Variable**
`initially` $in = \bot$	$in = m$ `unless`$_{\mu_1}$ $in = \bot$	`initially` $b_1 = \bot$
`initially` $mid = \bot$	`stable`$_{\mu_1}$ $in = \bot$	`constant`$_{\overline{\mu_1}}$ b_1
$mid = m$ `unless`$_{\overline{\mu_1}}$ $mid = \bot$	`stable`$_{\mu_1}$ $mid = m$	
`stable`$_{\overline{\mu_1}}$ $mid = \bot$	`stable`$_{\mu_1}$ $\lvert b_1 \rvert \leq N_1$	
`stable`$_{\overline{\mu_1}}$ $in = m$	`constant`$_{\mu_1}$ $\langle in \rangle \bullet b_1 \bullet \langle mid \rangle$	
	$\lvert b_1 \rvert < N_1$ `leadsto` $in = \bot$	
	$\lvert b_1 \rvert > 0$ `leadsto` $mid \neq \bot$	

Specification $\mathbf{R_2} \Rightarrow \mathbf{G_2}$

Rely $\mathbf{R_2}$	Guarantee $\mathbf{G_2}$	Aux.Variable
$\mathtt{initially}\ mid = \bot$	$mid = m\ \mathtt{unless}_{\mu_2}\ mid = \bot$	$\mathtt{initially}\ b_2 = \bot$
$\mathtt{initially}\ out = \bot$	$\mathtt{stable}_{\mu_2}\ mid = \bot$	$\mathtt{constant}_{\overline{\mu_2}}\ b_2$
$out = m\ \mathtt{unless}_{\overline{\mu_2}}\ out = \bot$	$\mathtt{stable}_{\mu_2}\ out = m$	
$\mathtt{stable}_{\overline{\mu_2}}\ out = \bot$	$\mathtt{stable}_{\mu_2}\ \lvert b_2 \rvert \leq N_2$	
$\mathtt{stable}_{\overline{\mu_2}}\ mid = m$	$\mathtt{constant}_{\mu_2}\ \langle mid \rangle \bullet b_2 \bullet \langle out \rangle$	
	$\lvert b_2 \rvert < N_2\ \mathtt{leadsto}\ mid = \bot$	
	$\lvert b_2 \rvert > 0\ \mathtt{leadsto}\ out \neq \bot$	

4.3 Hiding and Access Restrictions

The program *Buffer* is the parallel composition of $Buffer_1$ and $Buffer_2$ where the variable mid, initially empty, is made local.

$$Buffer \equiv (\text{initially}\ mid = \bot\ \text{in}\ Buffer_1 \| Buffer_2) \setminus \{mid\}$$

To cope with such hiding [7, 9, 19], it suffices to extend the rely condition with the hypotheses that the environment does not modify the local variables and that the program starts with correct initial values for its local variables.

$$\frac{P\ \text{sat}_\mu\ (\underline{R} \Rightarrow G)}{P \setminus \{v\}\ \text{sat}_\mu\ (R \Rightarrow G)}\ \underline{R} = R \cup \{\mathtt{constant}_{\overline{\mu}}\ v\}$$

$$\frac{P\ \text{sat}_\mu\ (\underline{R} \Rightarrow G)}{(\text{initially}\ v = e\ \text{in}\ P)\ \text{sat}_\mu\ (R \Rightarrow G)}\ \underline{R} = R \cup \{\mathtt{initially}\ v = e\}$$

We also need the information that $Buffer_1$ does not access the variable out and $Buffer_2$ does not access the variable in. To cope with such access restrictions [7, 19], it suffices to extend the guarantee condition with a `constant` formula stating that a program does not modify a variable that it does not access.

$$\frac{P\ \text{sat}_\mu\ (R \Rightarrow G)}{P\ \text{sat}_\mu\ (R \Rightarrow \underline{G})}\ v \notin Var(P),\ \underline{G} = G \cup \{\mathtt{constant}_\mu\ v\}$$

4.4 Composition of Mutually Dependent Specifications

Assuming $Buffer_1$ and $Buffer_2$ satisfy $R_1 \Rightarrow G_1$ and $R_2 \Rightarrow G_2$ respectively, we prove that *Buffer* satisfies $R \Rightarrow G$ provided that $N = N_1 + N_2 + 1$. The following rule is easily derived from the above parallel, hiding and access restrictions rules:

$$\frac{\begin{array}{lll} Buffer_1\ \text{sat}_{\mu_1}\ (R_1 \Rightarrow G_1) & \underline{R}, \underline{G_1^S}, \underline{G_2^S} \vdash R_1 & \underline{R}, \underline{G_1^S}, \underline{G_2^S} \vdash G^S \\ Buffer_2\ \text{sat}_{\mu_2}\ (R_2 \Rightarrow G_2) & \underline{R}, \underline{G_1^S}, \underline{G_2^S} \vdash R_2 & \underline{R}, \underline{G_1}, \underline{G_2} \vdash G \end{array}}{Buffer\ \text{sat}_\mu\ (R \Rightarrow G)}$$

where

$$\begin{array}{l} \underline{R} = R \cup \{\texttt{constant}_{\overline{\mu}}\, mid,\ \texttt{initially}\, mid = \bot\} \\ \underline{G_1} = G_1 \cup \{\texttt{constant}_{\mu_1}\, out\} \\ \underline{G_2} = G_2 \cup \{\texttt{constant}_{\mu_2}\, in\} \end{array}$$

The four proof obligations can be carried out in the Unity framework. Following [15], we use a refinement mapping technique [1, 2] to cope with auxiliary variables. In this case, the refinement mapping is simply $b = b_1 \bullet \langle mid \rangle \bullet b_2$.

First, we illustrate the union rule by proving:

$$\underline{R}, \underline{G_1^S}, \underline{G_2^S} \vdash \texttt{stable}_{\overline{\mu_1}}\, mid = \bot \tag{4}$$

which is a part of the proof obligation $\underline{R}, \underline{G_1^S}, \underline{G_2^S} \vdash R_1$. In addition to the union rule of Sect. 2, its proof uses the following constant rule:

$$\frac{\texttt{constant}_{\mu}\ e}{\texttt{stable}_{\mu}\ e = t}\ t \text{ is a static term}$$

$\underline{R}, \underline{G_1^S}, \underline{G_2^S} \vdash \texttt{stable}_{\overline{\mu_1}}\, mid = \bot$	
1. $\texttt{stable}_{\mu_2}\, mid = \bot$	by G_2^S
2. $\texttt{constant}_{\overline{\mu}}\, mid$	by $\underline{R}$
3. $\texttt{stable}_{\overline{\mu}}\, mid = \bot$	by 2, constant rule
4. $\texttt{stable}_{\mu_2 \cup \overline{\mu}}\, mid = \bot$	by 1, 3, union rule
5. $\texttt{stable}_{\overline{\mu_1}}\, mid = \bot$	by 4, $\overline{\mu_1} = \mu_2 \cup \overline{\mu}$

The last step of the proof uses $\overline{\mu_1} = \mu_2 \cup \overline{\mu}$. This equality captures the idea that the environment of $Buffer_1$ is exactly $Buffer_2$ plus the environment of $Buffer_1 || Buffer_2$. It highlights that the use of sets of agents rather than two agents eases the composition problem.

Then, we illustrate the composition of liveness requirements by proving:

$$\underline{R}, \underline{G_1}, \underline{G_2} \vdash |b| < N\ \texttt{leadsto}\ in = \bot \tag{5}$$

which is a part of the proof obligation $\underline{R}, \underline{G_1}, \underline{G_2} \vdash G$. Its proof requires the use of the implication and disjunction-transitivity rules:

$$\frac{\texttt{invariant}\ p \Rightarrow q}{p\ \texttt{leadsto}\ q} \qquad \frac{p\ \texttt{leadsto}\ q \vee r,\quad r\ \texttt{leadsto}\ s}{p\ \texttt{leadsto}\ q \vee s}$$

$\underline{R}, \underline{G_1}, \underline{G_2} \vdash \lvert b\rvert < N$ `leadsto` $in = \bot$	
1. `invariant` $\lvert b_1\rvert \leq N_1$	by G_1^S, rename (1)
2. `invariant` $\lvert b_2\rvert \leq N_2$	by G_2^S, rename (1)
3. `invariant` $\lvert b\rvert < N \Rightarrow \lvert b_1\rvert < N_1 \vee \lvert b_2\rvert < N_2 \vee mid = \bot$	see $(*)$
4. $\lvert b\rvert < N$ `leadsto` $\lvert b_1\rvert < N_1 \vee \lvert b_2\rvert < N_2 \vee mid = \bot$	by 3, implication rule
5. $\lvert b_2\rvert < N_2$ `leadsto` $mid = \bot$	by G_2
6. $\lvert b\rvert < N$ `leadsto` $\lvert b_1\rvert < N_1 \vee mid = \bot$	by 4, 5, disj.-trans.
7. $mid = \bot$ `leadsto` $\lvert b_1\rvert < N_1 \vee in = \bot$	by (4), G_1, rename (3)
8. $\lvert b\rvert < N$ `leadsto` $\lvert b_1\rvert < N_1 \vee in = \bot$	by 6, 7, disj.-trans.
9. $\lvert b_1\rvert < N_1$ `leadsto` $in = \bot$	by G_1
10. $\lvert b\rvert < N$ `leadsto` $in = \bot$	by 8, 9, disj.-trans.
$(*)$ by 1, 2, $b = b_1 \bullet \langle mid \rangle \bullet b_2,\ N = N_1 + N_2 + 1$	

Finally, we sketch the next deduction to illustrate the use of safety requirements when proving liveness requirements:

$$\underline{R}, \underline{G_1}, \underline{G_2} \vdash mid \neq \bot \wedge \lvert b_2\rvert = 0 \ \texttt{leadsto}\ out \neq \bot \qquad (6)$$

This result is used in the proof of $\underline{R}, \underline{G_1}, \underline{G_2} \vdash \lvert b\rvert > 0$ `leadsto` $out \neq \bot$. Steps 1 to 6 are given without further explanations; step 7 uses the progress-safety-progress (psp) rule of Sect. 2.

$\underline{R}, \underline{G_1}, \underline{G_2} \vdash mid \neq \bot \wedge \lvert b_2\rvert = 0$ `leadsto` $out \neq \bot$	
1. $\texttt{stable}_{\overline{\mu_2}}\ mid \neq \bot$	by $\underline{R}, G_1$, union rule
2. $mid \neq \bot\ \texttt{unless}_{\overline{\mu_2}}\ out \neq \bot \vee \lvert b_2\rvert > 0$	by 1, weakening rule
3. $\texttt{constant}_{\mu_2}\ \langle mid \rangle \bullet b_2 \bullet \langle out \rangle$	by G_2
4. $mid \neq \bot\ \texttt{unless}_{\mu_2}\ out \neq \bot \vee \lvert b_2\rvert > 0$	by 3, undetailed calculus
5. $\lvert b_2\rvert < N_2$ `leadsto` $mid = \bot$	by G_2
6. $\lvert b_2\rvert = 0$ `leadsto` $mid = \bot$	by 5
7. $mid \neq \bot \wedge \lvert b_2\rvert = 0$ `leadsto` $out \neq \bot \vee \lvert b_2\rvert > 0$	by 2, 4, 6, psp rule
8. $\lvert b_2\rvert > 0$ `leadsto` $out \neq \bot$	by G_2
9. $mid \neq \bot \wedge \lvert b_2\rvert = 0$ `leadsto` $out \neq \bot$	by 7, 8, disj.-trans.

5 Related Work

Unity. Compared with pure Unity specifications of concurrent objects [15, 17], our contribution lies in the ability to cope with *mutually dependent* rely-guarantee specifications. In the buffer example of [15, 17], mutually dependent specifications are systematically avoided by moving a part of the specification *outside* the rely-guarantee scheme. Despite this important difference, the way of reasoning about specifications is preserved: the proofs in Sect. 4 are a straightforward rewriting of similar proofs in [15]. More fundamentally, we must note that the composition principle, hence our work, is founded upon a compositional model of programs whereas the underlying computational model of [6] is not compositional.

Proof obligations: Jones' work. Based on Jones' earlier work [10], the parallel rules of [19, 20, 21] can be viewed as applications of the composition principle for terminating programs. Although carried out in another framework, the proof obligations can be related to ours:

- In [19, 21], the safety proof obligations look like $R \vee G_2^S \vdash R_1$. These cannot be expressed in our framework: no disjunction is allowed between Unity formulas. However, even if we write $R, G_2^S \vdash R_1$, there is an implicit disjunction between R and G_2^S because they are specifications over distinct sets of agents. More precisely, the disjunction is eventually made explicit in the proof of $R, G_2^S \vdash R_1$ when applying the union rule: the disjunction appears as the union of the sets of agents (see proof of (4) in Sect. 4).
- The liveness proof obligation in [10, 20] requires the construction of a dynamic invariant linking successive states in a behaviour; this binary relation must be preserved by both the environment and the system transitions. In the Unity framework, it basically corresponds to using the progress-safety-progress rule for `leadsto` (see proof of (6) in Sect. 4): its premises r $\texttt{unless}_{\overline{\mu}}$ b and r $\texttt{unless}_{\mu}$ b express that the associate binary relation must be preserved by both the environment agents and the system agents.

TLA rely-guarantee specifications. We contribute to the rely-guarantee paradigm by using explicit distinct subscripts to distinguish the rely and the guarantee parts of a specification. For example, the $\overline{\mu}$ subscripts indicate that the rely condition constrains the environment only. Similar syntactic restrictions appear in the Temporal Logic of Actions (TLA) for open systems [3, 12]: the disjunction $\mu \vee F$ restricts the transition formula F to environment transitions, and the disjunction $\overline{\mu} \vee F$ restricts the same formula to system transitions. In TLA, unprimed and primed formulas refer to the state before respectively after the transition; with this convention, the formula p $\texttt{unless}_{\mu}$ q corresponds to the binary state relation described by $\overline{\mu} \vee (p \wedge \neg q \Rightarrow p' \vee q')$ in TLA. Although they can be expressed in TLA, our specifications are not in TLA canonical form [3]: we specify a *conjunction of restrictions* on the system transitions instead of a *disjunction of allowed* system transitions.

Action-based specifications. Consequently, compared to TLA and other action-based specifications of concurrent objects [3, 9, 11, 18], the Unity approach preserves the *conjunctive character* of a specification: an omitted requirement can simply be added to the conjunction. For instance, the specification of a bounded buffer is obtained from the specification of an unbounded one simply by adding the formula $\texttt{stable}_{\mu}$ $|b|<N$. Adding this requirement in action-based specifications implies revising the definition of each action. Furthermore, invariant-looking properties such as $\texttt{constant}_{\mu}$ $in \bullet b \bullet out$ appear explicitly. As a drawback w.r.t. action-based specifications, we note that unrealizable specifications are not excluded when using `leadsto`; in action-based approaches, unrealizable specifications may be avoided by replacing the liveness requirements with suitable fairness requirements on the actions. Actually, the two approaches may appear at different stages of the development process: once the Unity specification is established, it may be refined until identifying the system actions becomes necessary. A possible

extension of this work is thus the development of proof rules, based on [6], to prove the validity of an action-based specification w.r.t. a Unity rely-guarantee specification. Then, the formal development process goes on, using established refinement method for action-based specifications, e.g. [11].

Temporal operators versus auxiliary variables. Applying the composition principle of [2, 4] requires specifications that can be interpreted in the proposed model. Another candidate specification language would be the compositional version of the linear time temporal logic [5]. Unfortunately, the more powerful operators of temporal logic raise the complexity of reasoning about specifications [18]. By choosing the Unity logic, we follow the alternative approach of *e.g.* [9, 12, 18, 19]: the specification language is simple and the necessary expressive power is obtained by using auxiliary variables. Actually, only `leadsto` is a temporal operator and temporal reasoning is then avoided whenever possible. As claimed in [12, 18], this approach yields more natural specifications: a specification is made of an initial condition (`initially`), a set of allowed transitions (`unless`), and a set of liveness requirements (`leadsto`).

6 Conclusion

In order to reuse Abadi and Lamport's results on mutually dependent rely-guarantee specifications, we have adapted the `unless` operator of Unity. By simple syntactic restrictions, we have obtained formulas that constrain either the system transitions or the environment transitions. Then, we have illustrated the approach on an example, by applying the rule to compose mutually dependent rely-guarantee specifications of concurrent buffers.

An advantage of the approach lies in keeping the Unity style of reasoning about specifications: since the language is simple (short formal description), it yields rather intuitive proof rules, hence workable specifications. However, as discussed in [2], reasoning about concurrent systems remains a lengthy task, because of detailed calculations. Even the simple example of concurrent buffers generates lengthy proofs. Redoing proofs in response to changes in the initial specification could thus be a problem when scaling the approach to real-size developments [16].

Acknowledgements

I am grateful to Pierre-Yves Schobbens, Michel Sintzoff, and Ketil Stølen for their valuable comments on earlier drafts of this paper. I also thank Mete Celitkin, Yves Ledru, Philippe Massonet, and Thanh Tung Nguyen for their helpful suggestions.

References

1. M. Abadi and L. Lamport, The Existence of Refinement Mappings, in *Proceedings of the 3rd Annual Symposium on Logic In Computer Science*, 1988, pp. 165-175.
2. M. Abadi and L. Lamport, Composing Specifications, in J.W. de Bakker, W.-P. de Roever, and G. Rozenberg eds., *Stepwise Refinement of Distributed Systems*, Springer-Verlag, 1990, LNCS 430, pp. 1-41.

3. M. Abadi and L. Lamport, An Old-Fashioned Recipe for Real Time, in J.W. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg eds., *Real Time: Theory in Practice*, Springer-Verlag, 1992, LNCS 600, pp 1-27.
4. M. Abadi and G.D. Plotkin, A Logical View of Composition and Refinement, in *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages*, 1991, pp 323-332.
5. H. Barringer, R. Kuiper, and A. Pnueli, Now you may Compose Temporal Logic Specifications, in *Proceedings of the 16th ACM Symposium on Theory of Computing*, 1984, pp. 51-63.
6. K.M. Chandy and J. Misra, Parallel Program Design: a Foundation, Addison-Wesley, 1988.
7. P. Collette, Semantic Rules to Compose Rely-Guarantee Specifications, Research Report RR 92-25, Université Catholique de Louvain, 1992, Belgium.
8. F. Dederichs, System and Environment: The Philosophers Revisited, Technical Report TUM-I9040, Institut für Informatik, Technische Universität München, 1990, Germany.
9. P. Grønning, T.Q. Nielsen and H.H. Lovengreen, Refinement and Composition of Transition-based Rely-Guarantee Specifications with Auxiliary Variables, in K.V. Nori and C.E. Veni Madhavan eds., *Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag, 1991, LNCS 472, pp 332-348.
10. C.B. Jones, Tentative Steps Towards a Development Method for Interfering Programs, in *ACM Transactions on Programming Languages And Systems*, 1983, Vol 5, 4, pp 596-619.
11. B. Jonsson, On Decomposing and Refining Specifications of Distributed Systems, in J.W. de Bakker, W-P. de Roever, and G. Rozenberg eds, *Stepwise Refinement of Distributed Systems*, Springer-Verlag, 1990, LNCS 430, pp. 261-385.
12. L. Lamport, The Temporal Logic of Actions, Research Report 57, Digital Equipment Corporation Systems Research Center, 1990.
13. Z. Manna and A. Pnueli, The Anchored Version of the Temporal Framework, in J.W. de Bakker, W.-P. de Roever, and G. Rozenberg eds., *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, Springer-Verlag, 1989, LNCS 354, pp. 201-284.
14. J. Misra and K.M. Chandy, Proofs of Networks of Processes, in *IEEE Transactions on Software Engineering*, 1981, Vol 7, 4, pp 417-426.
15. J. Misra, Specifying Concurrent Objects as Communicating Processes, in *Science of Computer Programming*, 1990, Vol 14, 2-3, pp. 159-184.
16. A. Pizzarello, An Industrial Experience in the use of UNITY, in J.P. Banâtre and D. Le Métayer eds., *Research Directions in High-Level Parallel Programming Languages*, Springer-Verlag, 1991, LNCS 574, pp 39-49.
17. A.K. Singh, Specification of Concurrent Objects Using Auxiliary Variables, in *Science of Computer Programming*, 1991, Vol 16, pp 49-88.
18. E.G. Stark, Proving Entailment Between Conceptual State Specifications, in *Theoretical Computer Science*, 1988, Vol 56, pp 135-154.
19. K. Stølen, A Method for the Development of Totally Correct Shared-State Parallel Programs, in J.C.M. Baeten and J.F. Groote eds., *Proceedings of Concur'91*, Springer-Verlag, 1991, LNCS 527, pp 510-525.
20. J.C.P. Woodcock and B. Dickinson, Using VDM with Rely and Guarantee-Conditions, in R. Bloomfield, L. Marshall and R. Jones eds., *Proceedings of VDM'88: The Way Ahead*, Springer-Verlag, 1988, LNCS 328, pp 434-458.
21. Q. Xu and H. Jifeng, A Theory of State-based Parallel Programming: Part I, in J. Morris ed., *4th BCS-FACS Refinement Workshop*, Springer-Verlag, 1991, pp 326-359.