A Rely-guarantee Discipline for Open Distributed Systems Design

Carlos H. C. Duarte [1]

Banco Nacional de Desenvolvimento Econômico e Social, Av. Chile 100, Centro, Rio de Janeiro, RJ, Brasil, 20001-970, e-mail: carlos.duarte@computer.org

Tom Maibaum

Department of Computer Science, King's College London, Strand, London, United Kingdom, WC2R 2LS, e-mail: tom@maibaum.org

Abstract

A number of authors have studied the design of distributed systems considering the existence of an environment over which little (if any) control is retained. Perhaps the most systematic of these studies suggest the use of rely and guarantee conditions that assert respectively what is assumed from the environment and what the system is committed to insure as long as the assumptions hold, a refinement of the pre and post-conditions adopted in sequential program design. We propose a new rely-guarantee discipline based on linear time future temporal connectives and show how it can be applied in the design of open distributed systems.

Keywords: Rely-guarantee discipline; Specification; Verification; Temporal logic; Formal methods; Software design; Software engineering.

1 Introduction

The design of distributed systems considering the existence of an *environment* over which little or no control is maintained has attracted increasing attention. As Abadi and Lamport have remarked [1], no system will exhibit the expected behaviour in the presence of a sufficiently hostile environment. So, by considering the existence of an environment, one may reduce the chance of leaving specification errors unnoticed. But the major benefit of such a refined design discipline for *open distributed systems* appears to be the achieved division of complexity: by hierarchically partitioning the universe into system and environment, one may obtain not only modular specifications and proofs but a compositional development process as well.

Perhaps the most systematic way of approaching the design of open distributed systems is to use *rely-guarantee conditions* as part of specifications or proofs, a clear refinement of pre and post-conditions adopted in sequential program design. Rely assertions express assumptions about the behaviour of the environment and guarantee assertions say what properties the system is committed to insure as long as all the assumptions hold. Providing this temporalised meaning, Chandy and Misra [4] solved the problem of circularity in composing complementary assertions about system and environment, although it was Pnueli [19] who later adopted for the first time temporal logic for this purpose. Jones [12] made their reading of these assertions more rigorous by establishing a connection with the potential interference between concurrent processes and by relating them to pre and post-conditions.

[1] Supported by CNPq, the Brazilian National Research Council, while developing part of this work.

Article published in Information Processing Letters 74 (2000) 55–63

Many other authors have also studied rely-guarantee design [3,5,20]. In general, these studies are based on specific programming formalisms which belong to a concrete level of abstraction. Also, rely-guarantee constructions are used throughout the entire design, e.g. as part of specifications. We feel that in this way, in contrast to the use of the same assertions only in the verification of some properties, a conflict is created with the usual practice in stepwise software development of viewing specifications as objects that resemble as a whole executable programs and may be realised in this form. There is some confusion about what it might mean to realise a rely-guarantee specification: one cannot implement such a specification correctly if it means that the operation itself must somehow guarantee the rely condition. Another central debate has been around the expressive power of the underlying logical system and, consequently, concerning how to develop some forms of syntactic reasoning about rely-guarantee constructions. The idea of Abadi and Lamport [1] is semantic, based on closures of future temporal formulas. Jonsson and Tsay [13] develop an analogous work, also based on closures, but at the syntactic level and in terms of past temporal connectives. Moszkowski [17] adopts fixpoints of future interval temporal logic formulas.

Here we attempt to address these issues using a propositional linear time logical system of objects inspired by the COMMUNITY language of Fiadeiro and Maibaum [10]. A problem related to this system was how to verify component properties taking into account interference from the environment. We deal with this issue proposing a conditional style of verification, which supports modular proofs through the use of derivable inference rules. We also clarify in this way the connection between the rely-guarantee discipline and the categorical approach to software development [8,10]. In particular, we focus on components (subsystems) of systems, which may contain many operations rather than individual operations *per se*. The idea is to assert properties of such components which may then be used in verifying properties of the system containing the component. We have evidence that these approach and discipline are not tied to the chosen logical system and, indeed, can be adapted to other formalisms and levels of abstraction. See [6,7] for examples on the use of a branching time logical system and the specification of both synchronous and asynchronous message passing.

The remainder of the paper is organised as follows: Section 2 introduces our temporal logical system; Section 3 presents our results; and Section 4 proposes as an example the design of a memory replication protocol for distributed systems. We conclude the paper commenting on related and future work.

2 Temporal Logical Preliminaries

We essentially adopt as a means of specifying and reasoning about open distributed systems the temporal logical system of objects introduced in [8]. This is, in effect, a more structured approach to software design if compared to early applications of temporal formalisms in this field [11,16]. Each component specification is written in terms of a language which is determined by a set of extra-logical symbols belonging to a signature:

Definition 1 (Signature) A *signature* $\Delta = (\Gamma, \mathcal{A})$ is a pair of disjoint and finite families of symbols such that:

- Γ is a set of *action symbols*;
- $\mathcal{A} = (\mathcal{A}_g, \mathcal{A}_l)$ is a pair of disjoint sets of flexible *attribute symbols*. The members of $\mathcal{A}_l$ denote *local attributes* whose changes are only effected by the specified component. The members of $\mathcal{A}_g$, denoting *global attributes*, are not logically constrained.

Action symbols denote the occurrence of instantaneous events. They may specify, e.g., the execution of a sequential program procedure or a distributed task. They are logically distinct from attribute symbols, which represent the current state of the specified part of reality. Attributes may be local, in which case

only the component can change their value, or global otherwise. This flexible interpretation of signature symbols (i.e. their meaning changes over time) is formalised as follows:

Definition 2 (Structure) An *interpretation structure* for a signature $\Delta = (\Gamma, (\mathcal{A}_g, \mathcal{A}_l))$ is a tuple $\theta = (\omega, G, A)$ where:

- ω is a linear discrete time structure;
- G maps each $g \in \Gamma$ to $G(g) \in \mathcal{P}(\omega)$;
- A maps each $f \in \mathcal{A}_g \cup \mathcal{A}_l$ to $A(f) \in \mathcal{P}(\omega)$ such that, for every $i \in \omega$, for every $f' \in \mathcal{A}_l$, either $\exists g' \in \Gamma \cdot i \in G(g')$ or $i \in A(f') \leftrightarrow i+1 \in A(f')$ (encapsulation of change).

The function G maps each action to the set of time points where it occurs. A maps attributes to the points where their value is TRUE. Local attributes are also constrained by the locality restriction, which is similar to the stuttering principle of Lamport [14].

We use formulas to write specifications and proofs. Apart form the classical propositional connectives $\rightarrow$ and $\neg$, we also consider that signature symbols define atomic formulas, the beginning of time denoted by **beg** defines a formula and the occurrence of a property q_2 until but not necessarily including the moment of occurrence of another property q_1, denoted by $(q_1)\mathbf{V}(q_2)$ (the strict strong until studied by Kamp), defines a formula as well.

Definition 3 (Formula) Given a signature $\Delta = (\Gamma, \mathcal{A})$, the set $F(\Delta)$ of *formulas* over Δ is defined below, provided that $h \in \Gamma \cup \mathcal{A}$ and $\{q_1, q_2\} \subseteq F(\Delta)$:

$$q ::= h \mid \mathbf{beg} \mid (q_1)\mathbf{V}(q_2) \mid q_1 \rightarrow q_2 \mid \neg q_1$$

We choose a floating interpretation in which temporal formulas are evaluated at particular time instants, instead of adopting the anchored version of Manna and Pnueli [16] where the initial moment plays a special role.

Definition 4 (Satisfaction) Given a signature $\Delta = (\Gamma, \mathcal{A})$, the *satisfaction* of a Δ-formula at instant $i \in \omega$ of a structure $\theta = (\omega, G, A)$ is defined as follows:

- $(\theta, i) \models h$ iff either $h \in \Gamma$ and $i \in G(h)$ or $h \in \mathcal{A}$ and $i \in A(h)$;
- $(\theta, i) \models \neg p$ iff $(\theta, i) \models p$ is not the case;
- $(\theta, i) \models p \rightarrow q$ iff $(\theta, i) \models p$ implies $(\theta, i) \models q$;
- $(\theta, i) \models \mathbf{beg}$ iff $i = 0$;
- $(\theta, i) \models p\mathbf{V}q$ iff there is $j \in \omega$ such that $i < j$, $(\theta, j) \models p$ and $(\theta, k) \models q$ for any $k \in \omega$ where $i < k < j$.

Other classical connectives such as $\bot$, $\top$, $\wedge$, $\vee$ and $\leftrightarrow$ are defined as usual. Some of the following definitions of temporal connectives are required by our design discipline:

- $p\mathbf{U}q \stackrel{\text{def}}{=} q \vee (p \wedge q\mathbf{V}p)$ (p happens until q does);
- $\mathbf{X}p \stackrel{\text{def}}{=} p\mathbf{V}\bot$ (p happens in the next instant);
- $\mathbf{F}p \stackrel{\text{def}}{=} \top\mathbf{U}p$ (p happens now or sometime in the future);
- $\mathbf{G}p \stackrel{\text{def}}{=} \neg\mathbf{F}(\neg p)$ (henceforth p happens);
- $p\mathbf{W}q \stackrel{\text{def}}{=} \mathbf{G}p \vee p\mathbf{U}q$ (p happens until q does, if this ever happens).

Specifications rather than formulas are the adopted units of modularisation in design. The designer proposes specifications to represent unconditionally part of reality and these are subsequently used as a basis for all the reasoning that may be necessary.

Definition 5 (Specification) A *specification* is a pair $\Phi = (\Delta, \Psi)$ where Δ is a signature and Ψ is a finite set of Δ-formulas (the *specification axioms*).

To support formal reasoning in an effective manner, we need a proof system consisting in a finite set of axiom schemas and inference rules. Since propositional linear future time logic is quite well-known, we omit the presentation of a proof system for the logic fragment based on the strict strong until connective, refering the reader to the second axiomatisation in [11]. The axiomatisation of **beg** through an axiom and a proof rule, which appears in [6], is also omitted. Of particular importance here is the logical axiom for capturing the encapsulation of change in local attributes, adapted from [8]. This is defined as: Given a specification $\Phi = ((\Gamma, (\mathcal{A}_g, \mathcal{A}_l)), \Psi)$, the locality of the respective attributes is written as follows:

$$loc(\Phi) \stackrel{\text{def}}{=} \bigvee_{g \in \Gamma} g \vee \bigwedge_{f \in \mathcal{A}_l} f \leftrightarrow \mathbf{X}f$$

This means that either an action happens or the values of the local attributes remain

unchanged. This temporal formula, seen as an axiom (which is also finite because the set of symbols in each signature is finite), is adopted as a part of our logical system. This notion of encapsulation gives an object-based character to our discipline of design.

We interconnect specifications based on morphisms that relate their languages:

Definition 6 (Signature Morphism) Given two signatures $\Delta_i = (\Gamma_i, (\mathcal{A}_{g_i}, \mathcal{A}_{l_i}))$, $1 \leq i \leq 2$, a *signature morphism* $\sigma : \Delta_1 \to \Delta_2$ is a pair of total functions (σ_g, σ_a), $\sigma_g : \Gamma_1 \to \Gamma_2$, $\sigma_a : \mathcal{A}_1 \to \mathcal{A}_2$ such that $\sigma_a(\mathcal{A}_{l_1}) \subseteq \mathcal{A}_{l_2}$. Signatures and their morphisms determine a category **Sig**.

This is inspired by the connection of COMMUNITY programs discussed in [10]. In particular, the restriction in this definition says that no local attribute of the component can become a global attribute of the environment.

The translation of formulas by morphisms is defined in a pointwise way, viewing them as sequences of logical and extra-logical symbols. Logical symbols are not affected by translation (since the logical language is global) but extra-logical symbols are translated according to each particular signature morphism definition. Now we can define specification morphisms to relate the behaviour of distinct components through their specifications:

Definition 7 (Specification Morphism) Given two specifications $\Phi_i = (\Delta_i, \Psi_i)$, $1 \leq i \leq 2$, a *specification morphism* $\phi : \Phi_1 \to \Phi_2$ is a signature morphism $\Delta_1 \to \Delta_2$ such that:
– for each $p \in \Psi_1$, there is a $\phi(p) \in \Psi_2$;
– $\Phi_2 \vdash \phi(loc(\Phi_1))$ (locality preservation).
Specifications and their morphisms determine a category **Spec**.

The second condition above is to guarantee that the locality of local attributes is preserved when they are considered as part of another specification. Note that this would not be the case otherwise because the locality axiom is a logical part of the formalism and does not belong to each specification unless explicitly stated, even though it always belongs to the respective generated theories.

3 Rely-Guarantee Design

Moving away from the traditional direct approach to design appears to be inevitable when the features of open distributed systems have to be treated. Isolated specifications establish only the local properties of each specified object. Interaction with the environment, say, is a global property which remains untreated in this way. A rely-guarantee discipline allows us to deal with global properties in an organised conditional manner.

Rely-guarantee design is based on the fact that the description of component behaviour can be relativised to take into account that of the environment, i.e., their interaction is described explicitly. Either in specifying or verifying some properties of a component, a rely clause defines a property related to the component which the environment is assumed to satisfy. A guarantee clause is also used to express the properties related to the environment which the component maintains provided that the assumption holds.

Since we wish to treat interaction with the environment while preserving the local discipline for specification and composition described so far, we introduce rely-guarantee constructions only for the purpose of verifying some properties. For a given specification Φ and finite sets of formulas *rely*, *pre*, *guar*, *post* based on Φ, we adopt assertions of the form

$$\{pre; rely\}\, \Phi\, \{guar; post\}$$

meaning that, whenever the pre-conditions *pre* are simultaneously established and all the assumptions *rely* are not violated unless the guarantees in *guar* and a post-condition in *post* are obtained first, the guarantees are not violated until and necessarily including the moment when the post-condition p is obtained, for all $p \in post$. Putting $J_P \stackrel{\text{def}}{=} \bigwedge\{p | p \in P\}$, the conjunction of all the properties in P, such rely-guarantee assertions (RG assertions, for short) are formalised as follows:

Definition 8 (RG Assertion) Given a specification $\Phi = (\Delta, \Psi)$ and $rely \cup pre \cup guar \cup$

$post \subseteq F(\Delta)$ such that these sets are finite, *rely-guarantee assertions* are defined below:

$$\{pre; rely\}\ \Phi\ \{guar; post\} \stackrel{\text{def}}{=}$$
$$\Phi \vdash \bigwedge_{p \in post} (J_{pre} \wedge (J_{rely})\mathbf{W}(p \wedge J_{guar}) \rightarrow (J_{guar})\mathbf{U}(p \wedge J_{guar}))$$

If compared to other rely-guarantee assertions in the literature, our definition is unusual. In [1,13], rely and pre-conditions appear all conjoined as an assumption formula, much in the way that guarantee and post-conditions are conjoined in a commitment formula. The separation adopted here emphasises the distinct role of each set of properties. A more pragmatic reason justifies the adoption of independently realisable post conditions. Most authors consider that, if the system behavior or specified operation terminates, this determines a definite state satisfying all the post-conditions [18]. A *wait* clause is sometimes introduced to express an invariant over the states of non-terminating computations [3]. Because we are dealing with open systems which are not required to terminate but eventually make true each of a number of properties, we prefer to adopt post-conditions in the way defined above.

Rely-guarantee constructions are interesting not only due to the additional discipline they introduce into dealing with global phenomena, but also because they may be decomposed and reused. A composition rule is normally proposed to achieve these effects in the verification of some properties. The following inference rule plays this role here:

Theorem 9 (Rule of Composition) Given a specification $\Phi = (\Delta, \Psi)$ and $\bigcup \{rely, guar\} \cup \{pre_i, post_i | 1 \leq i \leq 2\} \subseteq F(\Delta)$ such that the resulting set is finite, the proof rule below is derivable:

(COMP)
1. $\{pre_1; rely\}\ \Phi\ \{guar; post_1\}$
2. $\{pre_2; guar\}\ \Phi\ \{rely; post_2\}$

$\{pre_1 \cup pre_2; rely \cup guar\}$
Φ
$\{rely \cup guar; post_1 \cup post_2\}$

A stronger theorem taking into account an initialisation condition that supports additional presuppositions in assertions is proved in [6] based on the application of conventional axiom schemes and inference rules. Actually, Jonsson and Tsay [13] have remarked that composition rules such as the above are a consequence of the standard meaning of rely-guarantee assertions, which requires the guarantees to be true when each post condition is obtained, regardless of the truth of the assumptions then. Our Definition 8 captures this meaning including J_{guar} in the second argument of both **W** and **U**. Note that the first connective is used in the antecedent of the implication because therein the guarantees and the post condition cannot be assumed to happen and the definition of **W** ensures this. The until connective **U** adopted in the consequent of the same implication ensures that these formulas do eventually obtain.

The theorem above allows us to infer a more widely applicable composition rule as a corollary:

Corollary 10 (General Rule) Given a specification $\Phi = (\Delta, \Psi)$ and the finite set $pre \cup rely \cup guar \cup post \cup \{pre_i, rely_i, guar_i, post_i \mid 1 \leq i \leq 2\} \subseteq F(\Delta)$, provided that the following side-conditions are met, the inference rule below is derivable:

$$pre_1 \cup pre_2 \subseteq pre,$$
$$rely_2 \subseteq rely \cup guar_1,$$
$$guar \subseteq guar_1 \cup guar_2,$$
$$rely_1 \subseteq rely \cup guar_2,$$
$$post \subseteq post_1 \cup post_2,$$

(GCOMP)
1. $\{pre_1; rely_1\}\ \Phi\ \{guar_1; post_1\}$
2. $\{pre_2; rely_2\}\ \Phi\ \{guar_2; post_2\}$

$\{pre; rely\}\ \Phi\ \{guar; post\}$

The proof of this corollary is developed based on the application of **COMP** using the side conditions enumerated above and on the refinement of the given premises using the monotonicity of some temporal connectives. Note that this rule guarantees the composability of assertions, which are sometimes about the

Specification WRITER (or WR)
actions *up*!, *down*!
attributes *mem* : **local**
axioms
$up! \rightarrow \neg down!$ (1.1)
$up! \rightarrow \mathbf{X}(mem)$ (1.2)
$down! \rightarrow \mathbf{X}(\neg mem)$ (1.3)
End

Fig. 1. Specification of the writer

same object, but is not required to talk about the composability of specifications, since we may not want to describe distinct objects. These issues are illustrated in the next section. More elaborated rules may deal with hidden symbols which we do not feel are necessary here in view of the possibility of organising specifications in *design structures* supporting hidden features, as suggested in [9].

4 An Example

We adopt as an example the design of a memory replication protocol for a class of distributed systems. The main components of our system are the writer and reader modules. The writer module produces changes in a binary memory *mem* according to the commands *up*! and *down*! received from user processes. Each reader module has access to the writer memory as an external attribute of its environment and keeps an internal copy of this memory which is locally updated through the action *read* initially or when value changes in the writer memory occur. In addition, the state of each replica of the writer memory may be broadcast upon receipt of a *value*? request by the reader. The specifications of these objects appear in Figures 1 and 2.

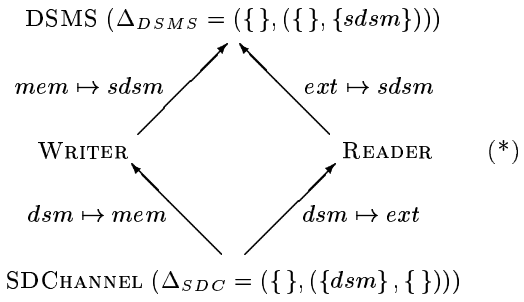
The first axiom in Figure 1 says that the two write actions *up*! and *down*! cannot occur at the same time. This means that we are specifying a single write/multiple read protocol for our distributed system. Note that, due to the logical encapsulation axiom of this specification, only the writer may effect any change in the original memory. The other

Specification READER (or RD)
actions *read*!, *value*?, *low*, *hi*
attributes *ext* : **global**; *int* : **local**
axioms
$\mathbf{beg} \rightarrow read!$ (2.1)
$read! \rightarrow \neg value?$ (2.2)
$ext \wedge \mathbf{X}(\neg ext) \rightarrow \mathbf{X}(read!)$ (2.3)
$\neg ext \wedge \mathbf{X}(ext) \rightarrow \mathbf{X}(read!)$ (2.4)
$read! \wedge ext \rightarrow \mathbf{X}(int)$ (2.5)
$read! \wedge \neg ext \rightarrow \mathbf{X}(\neg int)$ (2.6)
$read! \vee (int \leftrightarrow \mathbf{X}(int))$ (2.7)
$value? \wedge int \rightarrow \mathbf{X}(hi)$ (2.8)
$value? \wedge \neg int \rightarrow \mathbf{X}(low)$ (2.9)
End

Fig. 2. Specification of the reader

two axioms in the specification describe the changes in the memory value due to a user process write command. Concerning readers, 2.1 says that the writer memory is replicated in the beginning of time and 2.2 determines that updates and value queries are mutually exclusive. These prevent transient values of the memory replica *int* being read. Axioms 2.3 and 2.4 specify that *int* is updated whenever the value of the original memory *ext* changes. In addition, axioms 2.5, 2.6 and 2.7 specify how the writer memory is replicated and 2.8, 2.9 establish the pattern of answer to queries. Note that, in the case of a reader module, just the changes in the writer memory replica are encapsulated therein.

We formally interconnect reader and writer postulating the existence of a third specification of a single data channel with the global attribute symbol *dsm* only and the empty set of axioms. We also postulate the existence of two specification morphisms mapping *dsm* to *mem* and *ext* respectively. The specification DSMS of the whole system is obtained up to isomorphism by the *pushout* of the given morphisms. This requires the computation of an additional pair of morphisms and a new specification in which the symbol identified by the given morphisms is equalised and the following *pushout diagram* commutes:



This captures not only the fact that the data channel memory *dsm* is shared by reader and writer but also that it is seen as the writer local memory *mem* and the global reader memory *ext*, yielding the local attribute *sdsm* of the composed system. See [8] for an explanation of such categorical constructions.

An example of a rely-guarantee assertion based on DSMS is that, whenever the original memory is set up and the value of the local replica is not being queried, the replicated memory will eventually be positive when the query actually occurs, provided that meanwhile both the original memory is not set down and a query eventually happens. This assertion is formally written as follows:

Assertion MIRROR

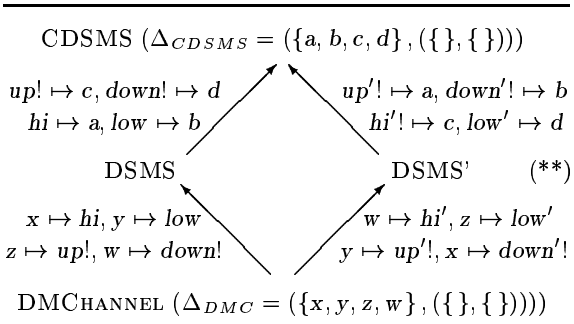
pre $up!, \neg value?$

rely $\neg down!, \mathbf{F}(value?)$

post $value? \wedge int$

To verify this assertion, first observe the connection between user process write commands and memory changes (1.2). In addition, the replica kept by the reader is always consistent with the writer memory (from our diagram and 2.1 to 2.6). The encapsulation of local reader attributes (loc(READER) plus 2.7) and the assumption of a query occurrence strictly after the pre-condition, without any other write request occurring meanwhile, show that the post-condition eventually obtains. Note that, whenever we omit a clause from an assertion, as above regarding the guarantees, we consider that the respective set of formulas is empty. Consequently, the logical clause value is true. It is not difficult to derive MIRROR using the adopted proof system.

In order to exemplify our composition rules, we need a more complex situation. Now we consider DSMS as a single object and compose two such objects connecting the production of query results to the occurrence of user process write requests. That is, we wish to establish two pairs of connections between *hi* and *down*! and between *low* and *up*! of two distinct distributed systems. In effect, the reader of each object controls the writer of the other. We illustrate the configuration of this complex system below:



Again, we postulate the existence of a specification and two morphisms, but this time the shared specification contains just two pairs of action symbols. This represents a pair of dual twisted message channels based on the synchronised actions of the respective objects. The specification of a complex DSMS is obtained up to isomorphism by a pushout construction as before.

The assertion MIRROR continues to be valid concerning each component of CDSMS. Using **GCOMP** and the morphisms connecting these components, we can conjoin two such assertions. Based on the way these components are connected, we can infer that, once each original memory is set up and no *value?* command is happening, provided that queries eventually happen, the value of these memories will be set down eventually. This is verified as follows, assuming a bit of familiarity with temporal logic:

1. $\{up!, \neg value?; \neg down!, \mathbf{F}(value?)\}$ CDSMS $\{; value? \wedge int\}$ — MIRROR, **
2. $\{up'!, \neg value'?; \neg down'!, \mathbf{F}(value'?)\}$ CDSMS $\{; value'? \wedge int'\}$ — MIRROR', **

3. $\{up!, \neg value?, up'!, \neg value'?;$ 1, 2
$\neg down!, \mathbf{F}(value?), \neg down'!, \mathbf{F}(value'?)\}$ **GCOMP**
CDSMS
$\{; value? \wedge int, value'? \wedge int'\}$
4. $up! \wedge \neg value? \wedge up'! \wedge \neg value'? \wedge$ **DEF-RG** 3
$\mathbf{F}(value?) \wedge \mathbf{F}(value'?) \rightarrow$ **DEF-W**, **DEF-F**
$\mathbf{F}(down!) \wedge \mathbf{F}(down'!)$ $\mathbf{G}(2.8)$, **MON-GF**, **MP**

The property above can also be explained by analogy with a misconfigured telecommunications system. Supposing that each DSMS represents a connection between two different equipment, the actions *up*! and *down*! have the intuitive meaning and *value*? stands for the temporary replacement of the handset, what the property says is that each call is eventually terminated by the disconnection of the other, if not before.

5 Concluding Remarks

In this paper we have proposed a new rely-guarantee discipline for the design of open distributed systems. This discipline is based on the strict strong until connective studied by Hans Kamp, which is logical or definable in most future linear time logics. We argued that rely-guarantee assertions should be introduced only in the verification process and derived two inference rules for composing such assertions. We showed through an example written in terms of a propositional temporal logical system of objects that our discipline is useful in practice. Now it is our intent to develop real case studies.

Many other authors have studied open systems design based on rely-guarantee constructions. Related work may be divided in process/model based formalisms [3,12,18] and logical ones [1,5,13,17]. In the latter work, many distinct levels of abstraction are normally discussed without a clear boundary, in part due to the influential view that implication coincides with refinement advocated by Abadi and Lamport [1]. In this latter category, only two kinds of assertions representing assumptions and commitments are usually considered. The aforementioned studies all permit extensive use of safety properties but just a few consider the occurrence of liveness properties as part of guarantee clauses. Both families of properties are treated uniformly here. In addition, the use of the connectives **W** and **U** relieve us from adopting the more demanding semantic closures and history variables in composition rules. Note that we have only been able to achieve these results due to the definition of the adopted formalism, which follows the open semantics proposed by Barringer [2] and considers that any number of actions belonging to a component or the environment may occur at each time.

The observation that rely-guarantee constructions may be introduced in the specification and verification processes is due to Pnueli [19]. We adopt just the latter view because we maintain in this way the practice in stepwise software development of considering specifications as objects that resemble and may be realised as executable programs. This may not be the case if rely-guarantee clauses are a standard part of specifications.

The choice above led us to the conclusion that rely-guarantee assertions used in the verification process only are orthogonal to the use of categorical techniques in design. Another possible way of adopting rely-guarantee constructions while resorting to a categorical approach is to internalise them as a distinguished part of each specification while reflecting the meaning of the respective properties in the definition of morphisms in the underlying category. The purpose of this new part of each specification would not be to provide realisable statements but to record decisions concerning each design. This appears to be an interesting further research subject. Preliminary results already appear in [15].

Acknowledgements: We wish to thank Carolyn Talcott for many discussions on open distributed systems.

References

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

[2] H. Barringer. The use of temporal logic in the compositional specification of concurrent systems. In A. Galton, editor, *Temporal Logics and their applications*, pages 53–90. Academic Press, 1987.

[3] A. Cau and P. Collete. Parallel composition of assumption-commitment specifications: A unifying approach for shared variable and distributed message passing concurrency. *Acta Informatica*, 33:153–176, 1996.

[4] K. M. Chandy and J. Misra. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[5] P. Collete. Composition of assumption-commitment specifications in a UNITY style. *Science of Computer Programming*, 23:107–125, 1994.

[6] C. H. C. Duarte. *Proof-Theoretic Foundations for the Design of Extensible Software Systems.* PhD thesis, Department of Computing, Imperial College, London, UK, 1998.

[7] C. H. C. Duarte. Proof-theoretic foundations for the design of actor systems. *Mathematical Structures in Computer Science*, 9(2):227–252, 1999.

[8] J. Fiadeiro and T. Maibaum. Temporal theories as modularisation units for concurrent systems specification. *Formal Aspects of Computing*, 4(3):239–272, 1992.

[9] J. Fiadeiro and T. Maibaum. Design structures for object-based systems. In S. Goldsack and S. Kent, editors, *Formal Aspects of Object-Oriented Systems.* Prentice Hall, 1994.

[10] J. Fiadeiro and T. Maibaum. Categorical semantics of parallel program design. *Science of Computer Programming*, 28(2–3):111–138, April 1997.

[11] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proc. 7th ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM Press, 1980.

[12] C. B. Jones. Specification and design of (parallel) programs. In R. E. A. Mason, editor, *Information Processing 83: IFIP 9th World Congress*, pages 321–332. Elsevier, 1983.

[13] B. Jonsson and Y.-K. Tsay. Assumption/guarantee specifications in linear time temporal logic. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *Theory and Practice of Software Development (TAPSOFT'95)*, volume 915 of *Lecture Notes in Computer Science*, pages 262–276. Springer-Verlag, 1995.

[14] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[15] A. Lopes and J. Fiadeiro. Preservation and reflection in specification. In M. Johnson, editor, *Proc. 6th International Conference on Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 380–394. Springer-Verlag, 1997.

[16] Z. Manna and A. Pnueli. The anchored version of the temporal framework. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 200–284. Springer-Verlag, 1989.

[17] B. Moszkowski. Using temporal fixpoints to compositionally reason about liveness. In H. Jifeng, J. Cooke, and P. Wallis, editors, *Proc. 7th BCS-FACS Refinement Workshop*, 1996.

[18] P. K. Pandya and M. Joseph. P-A logic: a compositional proof system for distributed systems. *Distributed Computing*, 5:37–54, 1991.

[19] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 123–144. Springer -Verlag, 1985.

[20] K. Stolen. A method for the development of totally correct shared-state parallel programs. In J. C. Baeten and J. F. Groote, editors, *Proc. 2th International Conference on Concurrency Theory (CONCUR'91)*, volume 527 of *Lecture Notes in Computer Science*, pages 510–525. Springer-Verlag, 1991.