Component-wise Formal Approach to Design Distributed Systems

I.S.W.B. Prasetya, S.D. Swierstra*, B. Widjaja†

Abstract

Giving a compositional proof for progress properties of distributed systems has always been problematic. Without compositionality, the correctness of a component cannot be verified in isolation (without full knowledge of its would-be partners), which in many contexts is a severe restriction. This paper presents an approach in which properties, and in particular progress properties, can be proven compositionally.

1 Introduction

Compositionality is a property of a programming logic that enables us to factorize a property of a program into properties of its components. It is useful as it reduces the number of proof steps (properties local to a component P only need to be verified against P rather than against the whole program) and leads to a better way to organize specifications and proofs (and hence it also helps to improve documentation and maintenance).

Compositionality is also an important aspect in the formal treatment of components in the component-based approach in software engineering, like when using COM, CORBA, or JavaBeans, because it enables separate development of components. This in it turns enhances the reusability of the components. However, having to develop a component separately, its developer usually has little knowledge nor control on the environment in which the component will be placed. With only partial knowledge on the environment, compositionality is the only way to separate the overall specification of the component into its own and its environment's *local* specifications. These can be tested and verified separately.

The assertional method is one of the earliest methods to reason about distributed programs [4, 3]. It has many variants, but essentially one decorates each location i in a program P with an assertion q_i to specify the possible states of the program when control reaches the location. Its correctness is proven by proving that all P's actions leading to location i establish the assertion q_i and

*Informatica Instituut, Universiteit Utrecht. Email: `wishnu@cs.uu.nl`, `doaitse.cs.uu.nl`.
†Fakultas Ilmu Komputer, Universitas Indonesia. Email: `bela@cs.ui.ac.id`

that P's partners preserve it. The method is compositional, but is based on partial correctness. Hence we cannot, within the logic, show that the program will actually progress from one location to another. To deal with progress in concurrent systems we need some form of temporal logic, like Manna and Pnueli's Linear Temporal Logic (LTL) [6], Lamport's Temporal Logic of Actions (TLA) [5], or Chandy and Misra's UNITY [1].

Early theories on composition of temporal properties [1, 7] actually believe that progress in general is not compositional. Later work such as [2] shows that this is the case. [2] specifies a component P by a pair (A, C) of *assume-guarantee* properties, stating P's commitment to behave as C when operating in any context that meets assumption A. The approach is compositional, but creates another problem. Progress is expressed in terms of the combined execution of a component and its environment. Eventually of course, we must reduce this to local properties of either the component or its environment. This reduction is unfortunately just as uncompositional as in [1]. This sounds like a paradox, and indeed it is. [2] is right in its view that we can always compose properties. The problem is in the amount of additional information needed to make a set of properties compose in the way we want. In [1, 2], if P's progress from p to q is to be preserved by its environment, the environment must respect all P's intermediate progress, and therefore this information must be maintained in P's specification. In many cases this is too much to ask.

A 'more compositional' approach would attempt to reduce the required extra information, though at the expense of setting more constraining requirements on the components. For example, [11] proposes a stronger kind of progress with nice composition properties, but unfortunately is limited in its use. [13] proposes to use an abstract component to specify a program's environment. Abstract components compose well, but require the actual environment of a component to refine its abstract environment in an action-wise way, which poses a quite severe restriction.

Our work takes a different approach. We observe that many techniques for synchronizing components are based on alternately scheduling one party to temporarily suspend interference on a certain data segment to allow others to progress. By specifying the moment on which this non-interference period starts, its duration, the data segment being released from interference, and the way the non-interference period may end we can infer how components influence each other. An earlier form of this idea first appears in an unpublished work of A.K. Singh [14] –the theory still suffers from lack of proof heuristics, formality, and consistency [8], although later work by Prasetya [9] manages to cast it in a formal and consistent framework. Both [14, 9] only capture temporary non-interference on a fixed set of variables shared by two components. This is a severe limitation because in general the uninterfered portion of the data segment changes at the run-time.

Our theory has also been mechanically checked in the theorem prover HOL, so it should be quite trustworthy. A full listing of the theory and its HOL proof is available at `www.cs.uu.nl/~wishnu`.

Paper Content
Section 2 illustrates the usefulness of compositionality and the issues that confront us there. Section 3 presents our theory for composition. Section 4 gives some simple examples to show how to employ theory. Section 5 presents a formal model, and Section 6 gives some conclusions.

2 Compositionality

To illustrate the usefulness of compositionality and the issues that challenge us, let us start with an example. Consider a component called *Remote Terminal Unit* (RTU) used to monitor the flow and water level in a portion of a river and to control a set of gates guarding the flow into and from that portion of river. The RTU does not have intelligence of its own to decide on the best gates' configuration at any given moment, and this decision is delegated to an external component called *Control Centre* (CC).

Suppose now we want to develop the CC, and are allowed to specify the minimal requirement of the RTU. We do not make RTU. We also do not want to base our design on a fixed RTU for two good reasons: (1) to broaden our market, and (2) to better shield our CC from being affected by implementation changes by RTU developers. We start by writing down its specifications. Since CC is intended to work in conjunction with RTU, we start with a specification of the joint behavior of $\texttt{CC}\|\texttt{RTU}$ where $\|$ stands for parallel composition, as in:

$$
\begin{array}{l}
\texttt{CC}\|\texttt{RTU} \\
\vdash \\
\mathsf{RTU}_{\mathsf{rdyToRep}} \wedge \mathsf{gates}, \mathsf{sensors} = X, Y \\
\mapsto \\
\mathsf{gates} = (\texttt{Execute} \circ \texttt{Decide} \circ \texttt{Pack})(X, Y)
\end{array}
\tag{1}
$$

where $\mathsf{RTU}_{\mathsf{rdyToRep}}$ is a location in RTU indicating that it is prepared to send its report (in the 'dummy' RTU in Figure 1 this corresponds to `r4`). The specification states that whenever `RTU` is ready to report its gate and sensor data, then eventually the gate will be properly reconfigured based on the data the `RTU` says it wants to report.

In the mean time we manage to come up with a prototype of CC which is displayed in Figure 1. An RTU is also displayed there, but this is just a dummy because we do not want to fix the RTU. The c_i and r_i in the code are labels indicating program locations. As we will refer to this examples later, we will first explain how the protocol between CC and RTU works. This is not essential for understanding the theory, and can be skipped if the reader wishes.

The Protocol
The communication between RTU and CC is assumed to be error free and synchronous. They use '*ports*' to communicate. There are two: `c` and `r`. Port c is used by CC to send commands to RTU, and port r is used by RTU to send status reports to CC. A message can

```
CC ::                                  |  RTU ::
[OUT   c,r : Int                       |  [OUT    c,r : Int
 LOCAL report,newConf : Int            |   LOCAL  command,gates,sensors : Int
 [c0] c:=Inquiry ;                     |   WHILE true DO
      WHILE true DO                    |   [[r0] WAIT WHILE c=0 ;
      [[c1] WAIT WHILE r=0 ;           |    [r1] command,c := c,0 ;
       [c2] report,r := r,0 ;          |    [r2] gates := Execute command ;
       [c3] newConf  :=                |    [r3] ''sample sensors'' ;
            Decide report ;            |    [r4] r:=Pack(gates,sensors)
       [c4] c :=  newConf ]            |   ]
]                                      |  ]
```

Figure 1: The CC and the RTU in SPL

only be written to a port when it is 'empty'[1]. Conversely, a message can only be retrieved from a port if it is filled. To keep it simple, an empty port is represented by value 0 and real messages are assumed to be non-zero.

RTU listens to commands sent by CC. When it receives one it immediately executes it. Next, it samples the new state of the sensors. The new gates and sensors states are then packed and sent back to CC. CC begins by sending an `Inquiry` command to RTU to have it report its starting state to CC. Upon receiving a report from RTU, CC will decide on the next configuration of the gates, which is then sent to RTU.
□

Before releasing the product, surely we want to verify that our CC meets (1). There is one big problem: this is not possible. The specification is expressed as a property of `CC‖RTU`, so to verify it we need RTU's code too, which we do not have. To insist on its availability would mean to stick to certain RTU(s), and would cancel the previously mentioned advantages of not fixing the RTU.

To overcome the above problem we need so-called *composition laws* which enable us to compose specifications without having to know the code. Such laws take the following form:

$$\frac{P \text{ satisfies } A \ , \ Q \text{ satisfies } B}{P \| Q \text{ satisfies } C(A,B)} \tag{2}$$

where P and Q are components and A, B and C are properties, and C is somehow obtained from A and B. Applying the law in reverse direction would allow us to factorize (1) into local specifications of CC and RTU which can subsequently be verified separately.

[1] This is not directly visible from the program in Figure 1, but will follow from the invariant in Figure 3.

Composition of temporal properties, in particular progress properties, has always been a difficult issue. To show where the difficulty lies, consider the following law proposal:

$$\frac{P \vdash p \mapsto q \ , \ Q \vdash \mathsf{stable}\ p}{P[\!]Q \vdash p \mapsto q} \tag{3}$$

stating that if P guarantees progress from p to q and if Q maintains p then the progress holds in the composite. This would make a powerful law, but unfortunately it is not valid. On its way to q, P may pass some state outside p. Q would no longer be constrained and may block P's progress to q. Hence progress to q in the composite is not guaranteed. Is there any way to get around this? Well, consider this property of RTU in Figure 1:

$$\texttt{RTU} \vdash r0 \wedge (c = X) \wedge (c \neq 0) \mapsto r3 \wedge (\mathsf{gates} = \texttt{Execute}\ X) \tag{4}$$

where the predicate r_i means that the program is in location r_i. It states that when RTU is ready to receive a command, and one is present in port c, this will be executed and the result is the new gates configuration. To realize (4) RTU goes through several intermediate 'stages' (see RTU's code): from $r0 \wedge (c = X) \wedge (c \neq 0)$ it goes to $r1 \wedge (c = X) \wedge (c \neq 0)$ then to $r2 \wedge (\mathsf{command} = X)$, and then to $r3 \wedge (\mathsf{gates} = \texttt{Execute}\ X)$. To keep (4) in the composite $\texttt{CC}[\!]\texttt{RTU}$ we can strengthen the earlier law proposal by requiring CC to maintain *all* intermediate stages of (4). This would work, and the good thing is that the constraint on the CC side is almost the weakest possible. The bad thing is that a progress specification of a component must now explicitly mention 'extra information', namely all its intermediate stages. This is too cumbersome, and we do not always have that information available either.

In our approach we will require CC to temporarily cease interference on c while RTU completes (4). This is a stronger requirement than the one above. On the other hand the approach also reduces the amount of 'extra information' needed to compose progress specifications, which hopefully will result in a reasonable compromise.

3 A Theory of Composition

This section presents our composition theory. The following notational convention will be used:

Notation
Programs or components range over by P and Q. Predicates range over p, q, r, s, J, and I. Sets of variables range over V, W, and Z. We write $J \vdash p$ to mean $(\forall s :: J\ s \Rightarrow p\ s)$, meaning that under assumption J, p holds everywhere, in which s refer to the state of the system.
□

3.1 Predicate Confinement

Total functions f and g of the same type, are partially equal over subdomain V, written $f =_V g$, if $f\ x = g\ x$ for all $x \in V$. The set of all functions of f such that $f =_V g$ is denoted by $[g]_V$. A program state is a function from variables to values, so we can talk about partially equal states in the above sense. A state predicate is a function from state to **bool**, or equivalently a set of states. Given a predicate p and a set of variables V, we define p to be *confined* by V if it only constrains the value of variables in V. This can be formally defined as follows:

$$p\ \mathbf{conf}\ V\ =\ (\forall s.\ p\ s = ([s]_V \subseteq p)) \tag{5}$$

For example, $x > y + z$ is confined by $\{x, y, z\}$, but not by any set smaller than $\{x, y, z\}$. Notice also that a predicate which is confined by V cannot be falsified by a program only updating outside variables V. A predicate confined by V is also confined by a larger set. Predicate confinement is also closed under predicate operators. Formally:

$$V \subseteq W\ \wedge\ p\ \mathbf{conf}\ W\ \Rightarrow\ p\ \mathbf{conf}\ V \tag{6}$$

$$p, q\ \mathbf{conf}\ V\ \Rightarrow\ (p \vee q\ \mathbf{conf}\ V) \wedge (\neg p\ \mathbf{conf}\ V) \tag{7}$$

3.2 Capturing Temporary Non-interference

In many standard models for concurrent systems, programs (even a high level one) can be regarded as simply a set of actions where each action is simple enough to be assumed atomic and terminating. Concurrent actions are modelled to be executed in interleaving, which is helpful in simplifying the theory. We also use this model.

Let A be a property of a component P which would be sensitive to external interferences on variables in V. Obviously, composing P with with another component Q which does *not* interfere on V will preserve A. However, requiring Q's to never interfere on V will in many cases be too restrictive, and thus we look for some form of temporary non-interference. To express this, we write $p \vdash Q$ preserves $V|q$, stating that once p holds Q will maintain p and also preserve the value of variables in V, and this period of non-interference is ended by establishing q. Formally:

Definition 3.1 : PRESERVES
$J, p \vdash Q$ preserves $V|q\ =\ (\forall a, p' : a \in Q \wedge p'\ \mathbf{conf}\ V : \{J \wedge p \wedge p'\}\ a\ \{(p \wedge p') \vee q\})$
□

The a in the formula range over the actions of Q. We also add an extra parameter J which is intended to be a stable predicate of Q. This is just a predicate which cannot be falsified by any action of Q. The J parameter is not an essential addition, but helps in the presentation later. Since actions are assumed to be terminating, it does not matter whether the Hoare triple above means partial or total correctness. Here is an example of preserves property:

$$\mathsf{I}_{\mathtt{RTU}}, r \neq 0\ \vdash\ \mathtt{RTU}\ \mathsf{preserves}\ \{r\}|\mathsf{false} \tag{8}$$

stating that when $r \neq 0$ then RTU promises to preserve the value of r. Using the 'invariant' $\mathsf{I}_{\mathsf{RTU}}$ in Figure 3 it can be shown that this holds in the RTU of Figure 1.

There are some special cases of the preserves relation. The property $J, \mathsf{true} \vdash P$ preserves $V|q$ states that the only way P can interfere on V is by establishing q. The property $J, \mathsf{true} \vdash P$ preserves $V|\mathsf{false}$, which we also write as simply $J \vdash P$ preserves V, states that P never interferes with V. This is the same as saying that P has no write access to V, or in other words V only contains local variables of other components.

3.3 Composition

Suppose we have a component Q satisfying $J, p \vdash Q$ preserves $V|q$. This describes Q's temporary non-interference on variables in V. As noticed before, during this period of non-interference Q will obviously preserve P's behavior A if A is only sensitive to changes of variables in V. However, in many cases this is too much to require from P. Very often, this insensitivity needs to be restricted to changes which maintains a certain stable predicate or invariant J. This means that Q's updates outside V will not affect P's A as long as they result in states which are still within J.

The above is a very important observation, and in fact this is what we exploit to build up our theory. For a temporal property A let $P, V, J \vdash A$ means the conjunction of the following:

1. P satisfies property A.
2. J is stable in P. Formally :

$$P \vdash \mathsf{stable}\ J \ = \ (\forall a : a \in P : \{J\}\ a\ \{J\}) \tag{9}$$

3. The property A is insensitive to external changes on variables outside V, as long as these changes respect J. We also say that A is *confined* by V and J.

In the sequel we will only consider temporal properties which are expressed using the following relations. Their informal meaning is quite standard, but the formal definition of some of them cannot be given without referring to a formal semantics. This will be postponed until Section 5.

1. $P, V, J \vdash p$ unless q means that if at any moment during P's execution $J \wedge p$ holds, q may (but does not have to) hold eventually. Furthermore, p will remain to hold until q holds.
2. $P, V, J \vdash p \mapsto q$ means that if at any moment during P's execution $J \wedge p$ holds, q will hold eventually.

3. $P, V, J \vdash p$ unless q means that if at any moment during P's execution $J \wedge p$ holds, q will hold eventually. Furthermore, p will remain to hold until q holds. It can be defined as:

$$P, V, J \vdash p \text{ until } q \quad = \quad P, V, J \vdash p \mapsto q \ \wedge \ P, V, J \vdash p \text{ unless } q \qquad (10)$$

Moreover, properties of the form $P, V, J \vdash p \ Rel \ q$ also require both p and q to be confined by V.

As an example, consider again property (4) of the RTU in Figure 1. This would be a well formed formula in standard temporal logics e.g. [1, 5, 6], but not in ours. It misses the J and the V parameters. Consider now this specification:

$$\texttt{RTU}, Z, \mathsf{I}_{\mathsf{RTU}} \ \vdash \ r0 \wedge c = X \wedge c \neq 0 \ \mapsto \ r3 \wedge \mathsf{gates} = \texttt{Execute}\ X \qquad (11)$$

where $Z = \{\mathsf{pcR}, c, \mathsf{command}, \mathsf{gates}\}$ and pcR is RTU's program counter. This is a variable used by RTU to keep track of its point of control (the location it is at). So the 'predicate' $r0$ is actually equal to $\mathsf{pcR} = r0$.

The specification says more than (4), namely that the progress is also insensitive to changes in variables outside Z as long as these changes are kept within $\mathsf{I}_{\mathsf{RTU}}$. $\mathsf{I}_{\mathsf{RTU}}$ is some stable predicate. For example the $\mathsf{I}_{\mathsf{RTU}}$ in Figure 3 has been proven to hold. Its full meaning is not important now, but it does imply that while RTU is in $r1$ or $r2$ CC is waiting for a report from RTU (expressed by $\mathsf{CC}_{\mathsf{wait}}$) and hence can be expected not to interfere variables in Z (in particular c).

Note also that in properties of the form $P, V, J \vdash p \ Rel \ q$ we only require J to be stable rather than invariant. This spares us from having to explicitly specify the initial condition of a program. This is also useful when we want to analyze the behavior of a program beyond a certain point in its execution rather than all the way from the start. We will however retain the 'habit' to call J the 'invariant'. This has the same flavor of invariant subscript in the work of Sanders [12], but note we also use J to constrain interferences on variables outside V.

Let now Rel be either unless, until or $\mapsto$ and suppose that $P, V, J \vdash p \ Rel \ q$ holds. This property can be preserved in the composite $P[\!]Q$ if from the moment p holds Q ceases its interference on V, or in other words if $J, p \vdash Q$ preserves $V|$false. Well, we can also generalize this. Suppose now Q satisfies an arbitrary preservation constraint $J, a \vdash Q$ preserves $V|b$. If p holds, and it happens that a holds too, then we know that as long as a holds Q will preserve V at least until b holds. So, unless a prematurely goes down, or b prematurely goes up, during this time P can realize q. Formally:

Theorem 3.2 : (Generalized) Singh Law

$$\frac{\begin{array}{l} P, V, J \vdash p \ Rel \ q \\ Q \vdash \text{stable } J \\ a, b \ \textbf{conf} \ V \cup W \\ J, a \vdash Q \text{ preserves } V|b \end{array}}{P[\!]Q, V \cup W, J \vdash (p \wedge a) \ Rel \ ((q \wedge a) \vee \neg a \vee b)}$$

□

The law is originally due to Singh in his June-89 Notes on UNITY [14], but the above is significantly more expressive than the original one where the choice of V is fixed, namely the entire set of variables shared by P and Q. This is too restrictive. For example (8) states that RTU preserves r if $r \neq 0$. The property is needed to synchronize RTU and CC and is required to prove (1). However, c is also a shared variable of RTU and CC. In [14, 9] we would have to require that under condition $r \neq 0$ RTU preserves both r and c. This is unrealistic and would in fact put RTU in deadlock ($r \neq 0$ means RTU has put something on port r, so by its protocol it should now empty port c, but insisting it to preserve c would prevent it from emptying c).

The law above is a fundamental one that enables us to calculate in terms of $P, V, J \vdash p\ Rel\ q$ and preserves properties, but we need several more. These are displayed in Figure 2. The Locality Lift Law states that a property confined by V is also confined by a larger set. The J-Leftshift and J-Rightshift Laws are used to shift the stable predicate into and from the pre-condition (the p-argument) of $p\ Rel\ q$. The J-Strengthening Law states that we can strengthen the J-parameter. Weakening it is however not allowed. The J parameter of $\mapsto$ and until is also not disjunctive.

The next three theorems are corollaries of the Singh Law. Due to limited space the proof cannot be presented here.

The Transparency Law states that P's properties confined by a V consisting only of P's local variables is respected by any other component Q. The Until Composition Law states a condition under which an until property of a component can be preserved by a composition. The Scheduling Law is a more general version of Until Composition and is very useful. Suppose a component P can progress to q and that this progress is only sensitive to external changes to variables in V. Moreover, at the start of the progress P set a 'flag' a high to indicate its wish to progress uninterfered. This flag stays high until q is reached. On the other hand Q promises to suspend interference on V as long as a is high. The theorem justifies that in this case P's progress to q will be respected by Q.

In addition to those laws we of course still need the usual laws governing unless, until, and $\mapsto$. These can be found in the standard literature [1, 5, 6]. Due to limited space we cannot list them here.

4 Synthesizing Contextual Requirements

This section will briefly show how the laws from the previous section are used. Recall that in component based engineering components are developed separately. Separate development requires each component to be specified in the context of its environment without fixing the code of the environment. Without a composition theory such as presented in the previous section this would render separate verification of the component impossible.

Consider again CC's specification (1). Using the laws this can be verified without the code of RTU (we of course also need standard laws for temporal relations [1, 5, 6]). The result is shown in Theorem 4.1. As can be expected,

In the following theorems Rel represents either unless , until , or $\mapsto$.

Theorem 3.3 : Locality Lift

$$\frac{P, V, J \vdash p \; Rel \; q}{P, V, J \cup W \vdash p \; Rel \; q}$$

Theorem 3.4 : J-Strengthen

$$\frac{\begin{array}{l} P \vdash \mathsf{stable}\ J' \\ J' \vdash J \\ P, V, J \vdash p \; Rel \; q \end{array}}{P, J', V \vdash p \; Rel \; q}$$

Theorem 3.5 : J-Leftshift

$$\frac{\begin{array}{l} P, V, J \vdash (J' \wedge p) \; Rel \; q \\ P \vdash \mathsf{stable}\ J \wedge J' \\ p \; \mathbf{conf} \; V \end{array}}{P, V, J \wedge J' \vdash p \; Rel \; q}$$

Theorem 3.6 : J-Rightshift

$$\frac{\begin{array}{l} P, V, J \wedge J' \vdash p \; Rel \; q \\ P \vdash \mathsf{stable}\ J \\ J' \; \mathbf{conf} \; V \end{array}}{P, V, J \vdash (J' \wedge p) \; Rel \; (J' \wedge q)}$$

Theorem 3.7 : Transparency

$$\frac{\begin{array}{l} P, V, J \vdash p \; Rel \; q \\ Q \vdash \mathsf{stable}\ J \\ J \vdash Q \ \mathsf{preserves}\ V \end{array}}{P [\!] Q, V, J \vdash p \; Rel \; q}$$

Theorem 3.8 : Until Composition

$$\frac{\begin{array}{l} P, V, J \vdash p \ \mathsf{until}\ q \\ Q \vdash \mathsf{stable}\ J \\ J, p \vdash Q \ \mathsf{preserves}\ V | q \end{array}}{P [\!] Q, V, J \vdash p \ \mathsf{until}\ q}$$

Theorem 3.9 : Scheduling

$$\frac{\begin{array}{l} P, V, J \vdash a \wedge p \; Rel \; q \\ P, V, J \vdash a \ \mathsf{unless}\ q \\ Q \vdash \mathsf{stable}\ J \\ J, a \vdash Q \ \mathsf{preserves}\ V | \mathsf{false} \end{array}}{P [\!] Q, V, J \vdash a \wedge p \; Rel \; q}$$

Figure 2: Confinement and Composition Laws

the proof generates constraints on RTU. These are listed in the theorem.

To prove the theorem we will also need an invariant I_{CC} of CC. Finding out the right I_{CC} is not trivial, but this is also not what this paper is about. Figure 3 shows an I_{CC} which is sufficient to prove Theorem 4.1. Its invariance has been mechanically checked in HOL. A brief explanation on what it means is given in Subsection 4.1.

Notice that (1) is not a formula in our logic. For this to be the case we have to be more specific by specifying the confining set of variables (V), and the existence of some common invariant I that constrains RTU's interferences

outside V. Here is the new version:

$$\begin{array}{l} \texttt{CC}\|\texttt{RTU}, Z, I \\ \vdash \\ \mathsf{RTU}_{\mathsf{rdyToRep}} \wedge \mathsf{gates}, \mathsf{sensors} = X, Y \\ \mapsto \\ \mathsf{gates} = (\texttt{Execute} \circ \texttt{Decide} \circ \texttt{Pack})(X, Y) \end{array} \tag{12}$$

where Z contains all variables of CC and RTU and I is some combined invariant of CC and RTU. The theorem is given below, and the following notation will be used:

$P, V, J \vdash_{[p]} \ r \ Rel \ s$ abbreviates $P, V, J \vdash p \wedge r \ Rel \ p \wedge s$

Theorem 4.1 : Commitment-Requirement Specification of CC
Let CC be a program as in Figure 1 and let I_{CC} be CC's invariant as specified in Figure 3. For an arbitrary RTU the composition $\texttt{CC}\|\texttt{RTU}$ satisfies (12) provided RTU meets the following requirements:

$\mathsf{R}_{\mathsf{var}}$:		$\mathsf{pcR}, \mathsf{gates}, \mathsf{sensors} \in \mathbf{loc}(\texttt{RTU})$
R_0 :	CC	$\vdash \mathsf{stable}\ \mathsf{I}_{\mathsf{CC}} \wedge \mathsf{I}_{\mathsf{RTU}}$
R_1 :	RTU	$\vdash \mathsf{stable}\ \mathsf{I}_{\mathsf{CC}} \wedge \mathsf{I}_{\mathsf{RTU}}$
R_2 :	$\texttt{RTU}, \mathbf{loc}(\texttt{RTU})$,	$\mathsf{I}_{\mathsf{RTU}} \vdash_{[\mathsf{gates},\mathsf{sensors}=X,Y]} \mathsf{RTU}_{\mathsf{rdyToRep}} \mapsto \mathsf{RTU}_{\mathsf{wait}}$
R_3 :	$\texttt{RTU}, \mathbf{loc}(\texttt{RTU})$,	$\mathsf{I}_{\mathsf{RTU}} \vdash \mathsf{RTU}_{\mathsf{justGetCmd}} \wedge \mathsf{gates}, \mathsf{sensors} = X, Y \mapsto \mathsf{gates} = (\texttt{Execute} \circ \texttt{Decide} \circ \texttt{Pack})(X, Y)$
R_4 :	$\texttt{RTU}, \mathbf{loc}(\texttt{RTU})$	$\cup\{c\}, \mathsf{I}_{\mathsf{RTU}} \vdash_{[\mathsf{gates},\mathsf{sensors}=X,Y]} \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \ \mathsf{until}\ \mathsf{RTU}_{\mathsf{justGetCmd}}$
R_5 :	RTU	$\vdash \mathsf{stable}\ \mathsf{I}_{\mathsf{RTU}} \wedge \mathsf{RTU}_{\mathsf{wait}} \wedge \mathsf{gates}, \mathsf{sensors} = X, Y \wedge c = 0$
R_6 :	$\mathsf{I}_{\mathsf{RTU}}, r \neq 0$	$\vdash$ RTU preserves $\{r\}$|false
R_7 :	RTU	$\vdash \mathsf{stable}\ c = 0$
R_8 :	$\texttt{RTU}, \mathbf{loc}(\texttt{RTU})$,	$\mathsf{I}_{\mathsf{RTU}} \vdash \mathsf{true} \mapsto \mathsf{RTU}_{\mathsf{wait}}$

for some invariant $\mathsf{I}_{\mathsf{RTU}}$ of RTU. We assume the following locations in RTU: $\mathsf{RTU}_{\mathsf{rdyToRep}}$ is the location where RTU is ready to send its report to the port r; $\mathsf{RTU}_{\mathsf{wait}}$ is the location where the RTU waits for a command to arrive in port c; and $\mathsf{RTU}_{\mathsf{justGetCmd}}$ is the location just after RTU retrieves the command.
□

The list of requirements on RTU looks long, but they are not excessive. $\mathsf{R}_{\mathsf{var}}$ requires variables named pcR, gates, and sensors to exist in RTU. R_0 specifies that $\mathsf{I}_{\mathsf{CC}} \wedge \mathsf{I}_{\mathsf{RTU}}$ is invariant in CC. However, since I_{CC} is already an invariant of CC it actually says that CC should respect the invariance of $\mathsf{I}_{\mathsf{RTU}}$. The latter is an invariant of RTU and should be part of RTU's specification supplied by its developer. Similarly, R_1 states that RTU should respect the invariance of I_{CC}.

R_2 essentially says that after sending a report RTU eventually waits for a new command from CC. R_4 says that if a command arrives RTU will eventually

consume it. R_3 says that after consuming a command RTU will execute it and the effect is as prescribed by `Execute` released on the state of the gates and sensors as it was when the last report was sent. Then we also have a set of safety requirements. R_5 says that while waiting for a command from CC the RTU will continue to wait until a command arrives. R_6 says that once it puts a report in port r, RTU must not overwrite it before it is consumed by CC. R_7 says that RTU should not fill port c (quite obvious, since r is assumed to be its read port anyway). Finally we have one last progress requirement R_8 stating that RTU always returns to a state of waiting for a command from CC. Since the invariant I_{CC} states that RTU and CC cannot be both in the waiting state, R_8 is required to prove that CC does not have to wait forever for a report from RTU.

Due to limited space only a small fraction of the proof will be shown. The full theorem has however been mechanically verified with HOL and the code is available at `www.cs.uu.nl/~wishnu`. See also [10] for a more step-by-step guide on the proof. In general our approach was to partition a progress specification of a component in operation with its environment into pieces. Each piece is either: (1) implementable by internal progress (progress on local variables) of either the component or its environment, or (2) require synchronization. The first kind is reducible to local properties using the Transparency Law. The second kind really requires one party to temporarily cease interference on a certain set of variables while the other party continues.

Now let us have a look at some proof examples. Let $\mathbf{loc}(P)$ denotes the set of all local variables of P. Notice that if $P \neq Q$ then $J \vdash Q$ preserves $\mathbf{loc}(P)$. Consider again the specification 12. Since progress is transitive, this is implied by:

$$\mathsf{CC}[\!]\mathtt{RTU}, Z, I \;\vdash_{[\mathsf{gates},\mathsf{sensors}=X,Y]}\; \mathsf{RTU}_{\mathsf{rdyToRep}} \mapsto \mathsf{RTU}_{\mathsf{wait}} \tag{13}$$

$$\mathsf{CC}[\!]\mathtt{RTU}, Z, I \;\vdash_{[\mathsf{gates},\mathsf{sensors}=X,Y]}\; \mathsf{RTU}_{\mathsf{wait}} \mapsto \mathsf{RTU}_{\mathsf{justGetCmd}} \tag{14}$$

$$\begin{array}{lll} \mathsf{CC}[\!]\mathtt{RTU}, Z, I \;\vdash & (\mathsf{gates}, \mathsf{sensors} = X, Y) \wedge \mathsf{RTU}_{\mathsf{justGetCmd}} \\ & \mapsto \\ & \mathsf{gates} \in \mathtt{Execute}(X, (\mathtt{Decide} \circ \mathtt{Pack})(X, Y)) \end{array} \tag{15}$$

(13) and (15) would be good candidates to be implemented by RTU's internal progress. Look at the following derivation of (13):

$\mathsf{CC}[\!]\mathtt{RTU}, Z, I \;\vdash_{[\mathsf{gates},\mathsf{sensors}=X,Y]}\; \mathsf{RTU}_{\mathsf{rdyToRep}} \mapsto \mathsf{RTU}_{\mathsf{wait}}$

(1) $\Leftarrow$ { Locality Lift (Theorem 3.3), require $\mathsf{pcR}, \mathsf{gates}, \mathsf{sensors} \in \mathbf{loc}(\mathtt{RTU})$ }

$\mathsf{CC}[\!]\mathtt{RTU}, \mathbf{loc}(\mathtt{RTU}), I \;\vdash_{[\mathsf{gates},\mathsf{sensors}=X,Y]}\; \mathsf{RTU}_{\mathsf{rdyToRep}} \mapsto \mathsf{RTU}_{\mathsf{wait}}$

(2) $\Leftarrow$ { CC preserves $\mathbf{loc}(\mathtt{RTU})$, Transparency Law (Theorem 3.7) }

$\mathtt{RTU}, \mathbf{loc}(\mathtt{RTU}), I \;\vdash_{[\mathsf{gates},\mathsf{sensors}=X,Y]}\; \mathsf{RTU}_{\mathsf{rdyToRep}} \mapsto \mathsf{RTU}_{\mathsf{wait}}$

$\wedge$

$\mathsf{CC} \;\vdash\; \mathsf{stable}\ I$

(3) $\Leftarrow$ { J-strengthening (Theorem 3.4) }

$$\texttt{RTU}, \mathbf{loc}(\texttt{RTU}), \mathsf{I}_{\texttt{RTU}} \vdash_{[\mathsf{gates,sensors}=X,Y]} \mathsf{RTU}_{\mathsf{rdyToRep}} \mapsto \mathsf{RTU}_{\mathsf{wait}}$$
$$\wedge$$
$$\texttt{RTU} \vdash \mathsf{stable}\ I \ \wedge\ \mathsf{CC} \vdash \mathsf{stable}\ I$$

The decision to make (13) internal occurs in the first derivation step when we apply the Locality Lift Law to shrink the set of variables that confines (13) from Z to $\mathbf{loc}(\texttt{RTU})$. Subsequently the Transparency Law is used to 'coerce' the property to become RTU's local property. Because a component's local property should be provable using only the component's own invariant, we apply the J-Strengthening Law in the final step to weaken the invariant to RTU's own invariant.

Notice also that the above derivation also produces requirements $\mathsf{R}_{\mathsf{var}}$ and $\mathsf{R}_{0,1,2}$ of Theorem 4.1.

Property (14) is much harder to prove. When RTU is waiting for a command to arrive in port c, there are two possibilities. First, if a command has already arrived then we expect RTU to eventually retrieve it. Second, if the port is empty then we expect CC to eventually fill it. Here is a property capturing the first possibility:

$$\mathsf{CC}[\!]\texttt{RTU}, Z, I \vdash_{[\mathsf{gates,sensors}=X,Y]} \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \mapsto \mathsf{RTU}_{\mathsf{justGetCmd}} \qquad (16)$$

Achieving this would require synchronization. The RTU can consume the command in port c, but CC must temporarily suspend updates to c. Consider the following derivation:

$$\mathsf{CC}[\!]\texttt{RTU}, Z, I \vdash_{[\mathsf{gates,sensors}=X,Y]} \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \mapsto \mathsf{RTU}_{\mathsf{justGetCmd}}$$
(1) $\Leftarrow$ { Locality Lift (Theorem 3.3) }
$$\mathsf{CC}[\!]\texttt{RTU}, \mathbf{loc}(\texttt{RTU}) \cup \{c\}, I \vdash_{[\mathsf{gates,sensors}=X,Y]} \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \mapsto \mathsf{RTU}_{\mathsf{justGetCmd}}$$
(2) $\Leftarrow$ { definition of until }
$$\mathsf{CC}[\!]\texttt{RTU}, \mathbf{loc}(\texttt{RTU}) \cup \{c\}, I \vdash_{[\mathsf{gates,sensors}=X,Y]} \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \ \mathsf{until}\ \mathsf{RTU}_{\mathsf{justGetCmd}}$$
(3) $\Leftarrow$ { compositionality of until (Theorem 3.8) }
$$\texttt{RTU}, \mathbf{loc}(\texttt{RTU}) \cup \{c\}, I \vdash_{[\mathsf{gates,sensors}=X,Y]} \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \ \mathsf{until}\ \mathsf{RTU}_{\mathsf{justGetCmd}}$$
$$\wedge$$
$$\mathsf{CC} \vdash \mathsf{stable}\ I$$
$$\wedge$$
$$I, \quad (\mathsf{gates}, \mathsf{sensors} = X, Y \wedge \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0) \vdash \mathsf{CC\ preserves}\ \mathbf{loc}(\texttt{RTU}) \cup \{c\} | \mathsf{RTU}_{\mathsf{justGetCmd}}$$
(4) $\Leftarrow$ { see lemma below }
$$\texttt{RTU}, \mathbf{loc}(\texttt{RTU}) \cup \{c\}, I \vdash_{[\mathsf{gates,sensors}=X,Y]} \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \ \mathsf{until}\ \mathsf{RTU}_{\mathsf{justGetCmd}}$$
$$\wedge$$
$$\mathsf{CC} \vdash \mathsf{stable}\ I$$
$$\wedge$$
$$\mathsf{I}_{\mathsf{CC}}, \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \vdash \mathsf{CC\ preserves}\ \mathbf{loc}(\texttt{RTU}) \cup \{c\} | \mathsf{false}$$
(5) $\Leftarrow$ { J-strengthening (Theorem 3.4), CC does not write to $\mathbf{loc}(\texttt{RTU})$ }
$$\texttt{RTU}, \mathbf{loc}(\texttt{RTU}) \cup \{c\}, \mathsf{I}_{\texttt{RTU}} \vdash_{[\mathsf{gates,sensors}=X,Y]} \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \ \mathsf{until}\ \mathsf{RTU}_{\mathsf{justGetCmd}}$$
$$\wedge$$

$$\mathtt{RTU} \vdash \mathsf{stable}\ I \ \wedge\ \mathtt{CC} \vdash \mathsf{stable}\ I$$
$$\wedge$$
$$\mathsf{I}_{\mathtt{CC}}, \mathsf{RTU}_{\mathsf{wait}} \wedge c \neq 0 \vdash \mathtt{CC}\ \mathsf{preserves}\ \{c\}|\mathsf{false}$$

Again we use the Locality Lift Law in the first step to shrink the confining set of variables. So obviously it is decided that the progress in (16) can be confined to the **loc**(RTU) and c rather than to the entire Z. Next we strengthen the progress to an unless-kind of progress. This puts an extra requirement on RTU but in exchange we can use the Until Composition Law which is easier to deploy than the Singh Law. Applying the law produces a preserves condition on CC. Step 4 is a little bit complicated. It uses the following lemma:

$$\frac{J, p_1 \vdash P\ \mathsf{preserves}\ V|false \quad p_2\ \mathbf{conf}\ V}{J, p_1 \wedge p_2 \vdash P\ \mathsf{preserves}\ V|q}$$

The reader can verify it himself. We just use it to make the preservation condition on CC stronger, but simpler. The new preservation condition (after step 4) states that CC does not interfere on **loc**(RTU) and c as long as RTU is in $\mathsf{RTU}_{\mathsf{wait}}$ state and port c is filled. Since CC obviously does not write to RTU's local variable, the condition can be simplified further in step 5. The J-strengthening Law applies to step 5 for the same reason as it did in the derivation of (13): it weakens the invariant on the RTU's progress to RTU's own invariant.

The derivation above also produces requirement (R_4) of Theorem 4.1.

4.1 The System Invariant

For completeness sake the system invariant we come up with for CC (called $\mathsf{I}_{\mathtt{CC}}$) is displayed in Figure 3 as a conjunction smaller predicates. How we come up with it is beyond the scope of this paper, but its invariance has been verified in HOL. The total invariant of the system $\mathtt{CC}[\!]\mathtt{RTU}$ is $\mathsf{I}_{\mathtt{CC}} \wedge \mathsf{I}_{\mathtt{RTU}}$ where $\mathsf{I}_{\mathtt{RTU}}$ is left unspecified. Of course later on when a specific RTU is composed with CC then its $\mathsf{I}_{\mathtt{RTU}}$ has to be supplied too and its invariance with respect to CC will have to be checked. For the sake of completeness Figure 3 also shows an $\mathsf{I}_{\mathtt{RTU}}$ that matches the dummy RTU in Figure 1.

The two constants $\mathsf{CC}_{\mathsf{wait}}$ and $\mathsf{RTU}_{\mathsf{wait}}$ represent the locations where, respectively, CC and RTU are stuck in a waiting state for an incoming message. For CC this the location $c1$, and for our dummy RTU it is location $r0$.

The sub-invariants of $\mathsf{I}_{\mathtt{CC}}$ (and similarly those of $\mathsf{I}_{\mathtt{RTU}}$) can be grouped into three groups. The first sub-invariant encodes the range of the program locations. The second and third ones state some synchronization conditions: the second specifies the value the port r should contain when it is non-empty; the third states that when both ports c and r are empty, one of CC or RTU should be waiting while the other is actively processing (excluding dead-lock). The last four sub-invariants are 'local assertions', asserting some known properties of the state of CC at various program locations.

$$
\begin{array}{lll}
\mathsf{I}_{\mathsf{CC}}: & c0 \vee c1 \vee c2 \vee c3 \vee c4 & (17)\\
& r \neq 0 \Rightarrow r = \mathtt{Pack}(\mathsf{gates}, \mathsf{sensors}) \wedge c = 0 \wedge \mathsf{RTU}_{\mathsf{wait}} & \\
& c = r = 0 \Rightarrow \mathsf{CC}_{\mathsf{wait}} = \neg \mathsf{RTU}_{\mathsf{wait}} & \\
& c0 \;\Rightarrow\; \mathsf{newConf} = \mathtt{Inquiry} = (\mathtt{Decide} \circ \mathtt{Pack})(\mathsf{gates}, \mathsf{sensors}) & \\
& \qquad \wedge r = c = 0 \wedge \mathsf{RTU}_{\mathsf{wait}} & \\
& c2 \;\Rightarrow\; r = \mathtt{Pack}(\mathsf{gates}, \mathsf{sensors}) \wedge r \neq 0 \wedge c = 0 \wedge \mathsf{RTU}_{\mathsf{wait}} & \\
& c3 \;\Rightarrow\; \mathsf{report} = \mathtt{Pack}(\mathsf{gates}, \mathsf{sensors}) \wedge r = c = 0 \wedge \mathsf{RTU}_{\mathsf{wait}} & \\
& c4 \;\Rightarrow\; \mathsf{newConf} = (\mathtt{Decide} \circ \mathtt{Pack})(\mathsf{gates}, \mathsf{sensors}) \wedge r = c = 0 \wedge \mathsf{RTU}_{\mathsf{wait}} & \\
\mathsf{I}_{\mathsf{RTU}}: & r0 \vee r1 \vee r2 \vee r3 \vee r4 & (18)\\
& c \neq 0 \Rightarrow c = (\mathtt{Decide} \circ \mathtt{Pack})(\mathsf{gates}, \mathsf{sensors}) \wedge r = 0 \wedge \mathsf{CC}_{\mathsf{wait}} & \\
& c = r = 0 \Rightarrow \mathsf{CC}_{\mathsf{wait}} = \neg \mathsf{RTU}_{\mathsf{wait}} & \\
& r1 \;\Rightarrow\; c = (\mathtt{Decide} \circ \mathtt{Pack})(\mathsf{gates}, \mathsf{sensors}) \wedge c \neq 0 \wedge r = 0 \wedge \mathsf{CC}_{\mathsf{wait}} & \\
& r2 \;\Rightarrow\; \mathsf{command} = (\mathtt{Decide} \circ \mathtt{Pack})(\mathsf{gates}, \mathsf{sensors}) \wedge c = r = 0 \wedge \mathsf{CC}_{\mathsf{wait}} & \\
& r3 \;\Rightarrow\; c = r = 0 \wedge \mathsf{CC}_{\mathsf{wait}} & \\
& r4 \;\Rightarrow\; c = r = 0 \wedge \mathsf{CC}_{\mathsf{wait}} &
\end{array}
$$

Figure 3: The System Invariant of CC‖RTU.

5 Semantics Issue

This section gives a model for the composition theory presented in Section 3. The consistency of the theory within this model has been mechanically verified with HOL. The model is based on UNITY [1] though using another kind of temporal logic should also work. Essentially what we have to do is to provide the semantics of properties confinement. That is, we define the meaning of the parameter J and V in properties of the form $P, V, J \vdash A$ –this turned out to be not quite trivial. Next, we need to lift standard temporal properties laws to laws in terms of the $P, V, J \vdash A$ properties. This enables us to prove the validity of our composition laws. The lifted laws are also used during program development to replace the non-lifted ones.

A UNITY program is a set of terminating, atomic, and guarded actions – if the guard is false the action can still be executed (enabled) though with no effect (skip). An execution of a UNITY program is infinite, in which at each step an action is non-deterministically selected but the selection is weakly fair (no action can be indefinitely ignored). There are no primitive control structures (like sequencing, loop, or branch), though they can be programmed. Figure 4 shows the UNITY translation of the CC and RTU code in Figure 1. The actions of each component are listed, separated by `[]`. Parallel composition of components (between CC and RTU) is also denoted by `[]`.

In UNITY properties of programs are expressed in terms of unless, ensures, and $\mapsto$ operators. The intuitive meaning of unless and $\mapsto$ has been explained. Ensures expresses single action progress. Progress by $\mapsto$ is essentially a transitive and disjunctive composition of progress by ensures. We will have to modify the standard definitions of these operators to express properties confinement. For

```
CC :: [    if pcC=c0  ->  c,pcC := Inquiry,c1
        [] if pcC=c1 /\ r<>0  ->  pcC := c2
        [] if pcC=c2  ->  report,r,pcC := r,0,c3
        [] if pcC=c3  ->  newConf,pcC := Decide report, c4
        [] if pcC=c4  ->  c,pcC := newConf,c1              ]
[]
RTU:: [    if pcR=r0 /\ c<>0  ->  pcR := r1
        [] if pcR=r1  ->  command,c,pcR := c,0,r2
        [] if pcR=r2  ->  gates := Execute command
        [] if pcR=r3  ->  ``sample sensors''
        [] if pcR=r4  ->  r,pcR := Pack(gates,sensors), r0           ]
```

Figure 4: The CC and the RTU in UNITY

unless and ensures this is quite straightforward:

Definition 5.1 : UNLESS, AND ENSURES

$$\begin{array}{lcl} P,V,J \vdash p \text{ unless } q & = & P \vdash \text{stable } J \ \wedge \ p,q \ \mathbf{conf}\ V \ \wedge \\ & & (\forall a : a \in P : \{J \wedge p \wedge \neg q\}\ a\ \{p \vee q\}) \\ P,V,J \vdash p \text{ ensures } q & = & P,J,V \vdash p \text{ unless } q \ \wedge \\ & & (\exists a : a \in P : \{J \wedge p \wedge \neg q\}\ a\ \{q\}) \end{array}$$

□

Notice that it is the condition p,q **conf** V that imposes the confinement of the described behavior by V.

Defining confinement on progress by $\mapsto$ is more complicated. Here is an obvious candidate:

$$P,V,J \vdash p \mapsto q \ = \ (P \vdash J \wedge p \mapsto_{CM} q) \wedge (P \vdash \text{stable } J) \wedge p,q \ \mathbf{conf}\ V \quad (19)$$

where $\mapsto_{CM}$ is the standard $\mapsto$ as in Chandy and Misra's [1]. But this is too naive. The progress $p \mapsto q$ may pass through some intermediate stages. Since above we only require p and q to be confined by V, one of the intermediate stages may break this confinement. In UNITY $\mapsto$ is defined (inductively) as the least transitive and disjunctive closure of ensures. So, a $\mapsto$ property is constructed from elementary ensures properties. The latter define the intermediate stages of the $\mapsto$ property. To confine a $\mapsto$ property we must therefore also confine all its constituting ensures properties. This can be achieved by including the confinement condition within the inductive definition of $\mapsto$. Since the new ensures operator already contains the required confinement condition, we can define the new $\mapsto$ as the least transitive and disjunctive closure of the new ensures. Formally:

Definition 5.2 : LEADS-TO
For all P,V,J, the relation $(\lambda\ p\ q.\ P,V,J \vdash p \mapsto q)$ is defined as the smallest

Theorem 5.3 : PRE-STRENGTHENING AND POST-WEAKENING

$$\frac{P, V, J \vdash q \mapsto r, \quad J, V \vdash p \Rightarrow q, \quad J, V \vdash r \Rightarrow s}{P, V, J \vdash p \mapsto s}$$

The r-argument of unless or ensures can also be weakened as above, but the q-argument cannot be strengthened though it can be replaced by p if $J, V \vdash p = q$ (the infamous Substitution Law).

Theorem 5.4 : PROGRESS-SAFETY-PROGRESS (PSP)

$$\frac{P, V, J \vdash p \; Rel \; q, \quad P, V, J \vdash r \text{ unless } s}{P, V, J \vdash (p \wedge r) \; Rel \; (q \wedge r) \vee s} \quad Rel \text{ is either unless, ensures, or } \mapsto.$$

Theorem 5.5 : UNLESS AND ENSURES COMPOSITIONALITY

$$\frac{P, V, J \vdash p \text{ unless } q, \quad Q, V, J \vdash p \text{ unless } q}{P[\!]Q, V, J \vdash p \text{ unless } q} \qquad \frac{P, V, J \vdash p \text{ ensures } q, \quad Q, V, J \vdash p \text{ unless } q}{P[\!]Q, V, J \vdash p \text{ ensures } q}$$

Figure 5: Some standard UNITY laws expressed in the new logic.

relation $\rightarrow$ satisfying:

Lifting:	if $P, V, J \vdash p$ ensures q then $p \rightarrow q$.
Transitivity:	if $p \rightarrow q$ and $q \rightarrow r$ then $p \rightarrow r$.
Disjunctivity:	let W be non-empty; if for all $i \in W$ we have $p_i \rightarrow q$ then $(\exists i : i \in W : p_i) \rightarrow q$.

□

All UNITY laws in [1] hold under this new definition, although as expected they look slightly different. Just for illustration, some are displayed in Figure 5.

6 Conclusion

We presented a theory of composition based on capturing a component's temporary non-interference properties to infer how it influences other components' temporal properties. The consistency of the theory has been mechanically verified with respect to UNITY model. Having UNITY as model, the theory also inherits the usual UNITY laws to deal with non-compositional aspects of temporal properties. The theory itself is not restricted to UNITY, and with proper definition of properties confinement can be built on top of other temporal logics.

When used to factorize a specification of a composite $P[\!]Q$, the theory would typically result in stronger non-interference constraints than for example the approaches taken in [1, 2]. However, in exchange when specifying progress we do not need to keep track of its intermediate stages which in [1, 2] are needed for composing progress properties. We consider this a great advantage, since carrying around the intermediate stages is cumbersome, and moreover, the infor-

mation is not always available. Despite the restriction, we believe our approach to be useful based on the observation that many synchronization techniques implement synchronization by alternately schedule one party to suspend its interference while allowing the others to continue.

References

[1] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison-Wesley Publishing Company, Inc., 1988.

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

[3] M. Hulst. *Compositional Verification of Parallel Programs using Epitemic Logic and Abstract Assertional Languages*. PhD thesis, Dept. of Comp. Science, Utrecht University, 1995.

[4] L. Lamport. An assertional correctness proof of a distributed algoritm. *Science of Programming*, 2:175–206, 1982.

[5] L. Lamport. A temportal logic of actions. Technical Report 57, Digital Systems Research Center, April 1990.

[6] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Safety*. Springer-Verlag, 1995.

[7] J. Misra. Notes on UNITY 17-90: Preserving progress under program composition. Downloadable from `www.cs.utexas.edu/users/psp`, July 1990.

[8] I.S.W.B. Prasetya. Formalization of variables access constraints to support compositionality of liveness properties. In J.J. Joyce and C.J.H. Seger, editors, *LNCS 780: Higher Order Logic Theorem Proving and Its Applications*, pages 324–337. Springer-Verlag, 1993.

[9] I.S.W.B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Dept. of Comp. Science, Utrecht University, 1995.

[10] I.S.W.B. Prasetya, S.D. Swierstra, and B. Widjaja. Component-wise formal approach to design distributed systems (original). Draft. Availble at `www.cs.uu.nl/people/wishnu`. This is the original version, containing a complete proof matching the HOL proof as available at the same web-site., 1999.

[11] Udink. R. *Program Refinement in UNITY-like Environments*. PhD thesis, Utrecht University, 1995.

[12] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[13] N. Shankar. Lazy compositional verfication. Available at `www.csl.sri.com/fm-papers.html`., 1999.

[14] A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989.