A Theory for Composing Distributed Components, Based on Temporary Interference

I.S.W.B. Prasetya   T.E.J. Vos*   S.D. Swierstra†   B. Widjaja‡

Abstract

Compositionality provides the foundation of software modularity, re-usability and separate verification of software components. One of the known difficulties, when separately verifying components, is producing compositional proofs for progress properties of distributed systems. This paper presents a composition theory which is based on reasoning about temporary interference. The theory does not aim to be general, as existing standard theories do. In exchange for being restrictive, it becomes more powerful. The theory is also axiomatic: it captures aspects relevant for composition in a direct and clean way, resulting in a theory which is easy to understand. Presently, the theory only deals with components that synchronize by mutual exclusion, though it is in principle possible to extend it. The theory has been mechanically verified with respect to a UNITY semantics.

*Instituto Tecnológico de Informática, Universidad Politécnica de Valencia. email: tanja@iti.upv.es

†Informatica Instituut, Universiteit Utrecht. email: {wishnu,doaitse}@cs.uu.nl.

‡Fakultas Ilmu Komputer, Universitas Indonesia. email: bela@cs.ui.ac.id. Part of: Graduate Team Research Grant Batch III, University Research for Graduate Education (URGE) Project

1   Introduction

Preserving progress properties in parallel composition is problematic [5, 3]. Suppose $P \vdash p \mapsto q$ means: if p holds during an execution of P, eventually q will hold. Moreover, let $P \vdash p \text{ unless } q$ mean: if p holds during an execution of P, it will continue to hold at least until q holds. Then following law looks plausible:

$$\frac{P \vdash p \mapsto q \quad Q \vdash p \text{ unless } q}{P[\![Q \vdash p \mapsto q}$$ (1)

It states that P's progress $p \mapsto q$ is preserved in $P[\![Q$, if Q can maintain p at least until q holds. This would make a powerful law, but unfortunately it is not valid. On its way to q, P may move to some state outside p. When this happens Q is no longer be constrained and may obstruct P's progress to q.

A *theory of composition* is a theory stating how components' properties can be combined to a property of a system, or conversely, how a property of a system can be broken down into properties of its components. With such a theory we will be able to find which property B is required from program Q in order to preserve P's progress $p \mapsto q$ in $P[\![Q$. There are already several general theories, such as [1, 2, 7, 8, 5, 6]. In this respect, our theory does not offer anything new. It does offer a new approach, which can be useful. General theories, will give us the weakest B. The problem with choosing a weak B is that it is also less constructive (or under specified), giving less information to the implementator as to how the property can be implemented. Other theories, such as [4] will give us a stronger B. The problems with a too strong B is that it makes component Q less re-usable (or: it leaves less choices when selecting a component Q for the composed system $P[\![Q$). Our theory does not aim to be general. Instead, the idea is to obtain a B which is not overly restrictive and not too under specified. For comparison with existing theories, see [10].

2   Our Solution

Our theory is inspired by a composition law proposed by Singh in 1989 in his note on UNITY. The law is unfortunately unsound, but the idea it conveys is very useful. Consider a distributed system S consisting of a number of component processes and shared variables. We will often use the term *component* to mean 'component process' of a system and the term *common resource* to mean 'shared variable'. In a system where these components synchronize by mutual exclusion, the components alternately interfere with, or refrain interference on common resources. Consider a program P that requires a common resource x to achieve a certain goal, i.e. while working towards its goal, P interferes on x. Typically this inference is *limited* and *temporary*, meaning that in order to realize the goal, x is may be the only common resource P requires, and moreover once the goal is reached P will no longer need x. Now, program P will be able to reach its goal, if the environment synchronizes by temporarily stalling its interference on x, or in other words, if the environments synchronizes by limited and temporary *non-interference* on x. This is precisely what

our theory captures, the preservation of properties by scheduling interference and non-interference. We will give UNITY style operators to express temporary interference and non-interference.

2.1 Execution Model

A program is modeled as a set of atomic and terminating actions, and an execution of such a program is an infinite and interleaved execution of its actions. In each step some action is non-deterministically selected and executed. Finite stuttering is allowed in the executions. We assume that the selection of actions is weakly fair, meaning that an action which is continually enabled can not be ignored forever. Although strong fairness generally makes it easier to preserve a progress property, for various reasons we cannot always assume to have it.

2.2 Notation

P and Q denote programs; a, b, p, q, r and J are state predicates; α and β are actions (transitions); and V and W are sets of program variables. When specifying a law, as an abbreviation we write:

$$\frac{\vdash p \text{ unless } q \text{ using } V}{\vdash p \text{ unless } q \text{ using } W} \quad \text{instead of:} \quad \frac{P, J \vdash p \text{ unless } q \text{ using } V}{P, J \vdash p \text{ unless } q \text{ using } W}$$

hence omitting parameters which are constant in all entries of such a law.

2.3 Predicate Confinement

Predicates are used to specify a set of program states. A predicate p is *confined* by a set of variables V (written $p \text{ conf } V$) if p does not constrain the value of any variable outside V. For example, $x > y + z$ constrains the values of x, y and z, but does not constrain the values other variables. So, $x > y + z$ is *confined* by $\{x, y, z\}$. Confinement enables us to specify how sensitive a predicate is to interference. We write $p, q \text{ conf } V$ to abbreviate $p \text{ conf } V$ and $q \text{ conf } V$.

2.4 Temporary non-interference: The is preserved property

We introduce a relation is preserved to capture *temporary* non-interference by Q on a given set of variables V. The relation can be seen as extending UNITY [4] unless.

$$Q, J \vdash V \text{ is preserved when } p \text{ unless } q$$

means that for some stable predicate J, once p and J hold, Q will (1) maintain p and (2) refrain from interfering on variables in V (hence maintaining the values of those variables) at least until q holds. Note that the specified non-interference is only *temporary* (until q holds), and its scope is also *limited* (only on V). The need for parameter J will be explained in Subsection 2.6. For now it is sufficient to know that it is intended to be a stable predicate of Q:

Definition 2.1 : STABLE AND PRESERVES

$$Q \vdash \mathsf{stable}\ J \ \stackrel{d}{=}\ (\forall \alpha : \alpha \in Q : \{J\}\ a\ \{J\})$$

$$Q, J \vdash V \text{ is preserved when } p \text{ unless } q$$
$$\stackrel{d}{=}$$
$$(\forall \alpha, p' : \alpha \in Q \wedge p'\ \mathbf{conf}\ V : \{J \wedge p \wedge p' \wedge \neg q\}\ \alpha\ \{(p \wedge p') \vee q\})\ \ \wedge\ \ \ Q \vdash \mathsf{stable}\ J$$

People often ask why we do not use the co operator of new-UNITY. The choice is a matter of taste. Basic operators of UNITY and new-UNITY can be expressed in terms of each other. We find the classical operators more intuitive.

Temporary non-interference has nice properties, e.g. a component that preserves a set of variables V also preserves any subset of V. Moreover, since is preserved is defined as an extension of unless, it shares many properties of unless [4]. For example, weakening the postcondition q to q', given $J \wedge q \Rightarrow q'$, is allowed. The when-predicate p, cannot, in general, be strengthened or weakened. However, from the definition of is preserved (2.1) we can deduce that strengthening p with p' is allowed as long as p' is confined by V. Weakening p is not allowed. See also [10].

2.5 Local Variables

A component's local variables cannot be accessed by other components. So, if we know that Q preserves V, then this preserved set can always be extended by P's local variables ($\mathbf{loc}(P)$). This is expressed by the following law:

Theorem 2.2 : LOCAL PRESERVATION

$$\frac{P \neq Q \quad W \subseteq \mathbf{loc}(P)}{Q \vdash V \text{ is preserved when } p \text{ unless } q\ =\ Q \vdash V \cup W \text{ is preserved when } p \text{ unless } q}$$

The following is a useful corollary:

Corollary 2.3 :

$$\frac{P \neq Q \quad p'\ \mathbf{conf}\ \mathbf{loc}(P) \quad Q \vdash V \text{ is preserved when } p \text{ unless } q}{Q \vdash V \text{ is preserved when } p \wedge p' \text{ unless } q}$$

2.6 Limited interference: The using clause

We will now formalize the dual notion of non-interference, namely *interference*. Consider a progress property $p \mapsto q$ of P. It is an interference property, because typically it cannot be realized without interfering with —or writing to— some variables. The interference is *limited* in the sense that realizing $p \mapsto q$ depends only on a certain set, say V, of variables. The interference is also *temporary*: it starts the moment p holds and is no longer necessary as soon as q holds. To express this we introduce the notation $P \vdash p \mapsto q$ using V. Note that this property will imply that P's progress only depends on what happens to V, and hence is insensitive to external interference on variables outside V. However, requiring P to tolerate all sorts of interference on the variables outside V is often unrealistic. To be able to relax the requirement on P, the J-parameter from

Section 2.4 is introduced. Now only interference that preserves the stability of J need to be tolerated by P.

The parameter J also serves another function. Because of the insensitivity constraint, p and q should not constrain the values of the variables outside V, unless they do it in a way which is consistent with J. In favor of simplicity, we decide to simply forbid p and q from constraining any variable outside V (i.e. p and q have to be confined by V). This means that J is then *the only* place where we can specify the dependency of the property $p \mapsto q$ on variables outside V.

In our theory we only consider temporal properties A of the form p *Rel* q where *Rel* is either unless, $\mapsto$, or until. We require A to be a 'first order' temporal property in the sense that p and q should be state predicates, and not other temporal properties. The compositionality of this kind of properties is easier to formulate, and in practice we can express a lot of interesting properties using them. Below, we will give informal definitions of the operators –UNITY-style formal definitions can be found in [10].

Definition 2.4 : Unless

$$P, J \vdash p \text{ unless } q \text{ using } V$$

means: (1) p and q are confined by V; (2) J is stable in P; and (3) in an execution of P that starts is a state satisfying J: if P's environment, that may change the values of variables outside V, maintains J, then we have that once p holds it will remain to hold until q holds. (Note: q is not guaranteed to occur.)

Definition 2.5 : Leads-to

$$P, J \vdash p \mapsto q \text{ using } V$$

means: (1) p and q are confined by V; (2) J is stable in P; and (3) in an execution of P that starts is a state satisfying J: if P's environment, that may change the values of variables outside V, maintains J, then we have that once p holds q will eventually hold.

Notice that we require J only to be a *stable* predicate, and not necessarily an *invariant* (i.e. a stable predicate which holds from the very beginning of the program). Under parallel composition, a stable predicate is often more useful. For example, a program P may not be able to establish J without the help of another program Q. Such a J cannot be invariant of P. However P may still be able to maintain J stable, which can be useful to, for example Q, for realizing a certain progress property. For more discussion about this, see [10].

Until is a useful derived operator, which is formally defined as follows:

Definition 2.6 : Until

$$P, J \vdash p \text{ until } q \text{ using } V \quad = \quad P, J \vdash p \mapsto q \text{ using } V \ \wedge \ P, J \vdash p \text{ unless } q \text{ using } V$$

Intuitively, it means that if at any moment during an execution of P, $J \wedge p$ holds, it will remain to hold until q holds, and moreover q will eventually hold.

2.7 Composition

Now, we will look at some general composition laws for arbitrary properties Rel, where Rel can be either unless, until or $\mapsto$. Suppose we have two components P and Q satisfying:

$$
\begin{array}{rcl}
P, J & \vdash & p \; Rel \; q \text{ using } V \\
Q, J & \vdash & V \text{ is preserved when } a \text{ unless } b
\end{array}
$$

If during an execution of $P\|Q$ the condition $J \wedge p \wedge a$ holds, we know that Q will suspend its interference on V. This period of non-interference may last long enough for P to go over to q, but we cannot be totally sure about this. There are two scenarios that can cause the period of non-interference to end prematurely: (1) P destroys a, or (2) Q ends its non-interference by making b true. In case one of these scenarios takes place, there is nothing we can say about what will happen to $p \; Rel \; q$. However, we know that either $\neg a$ or b should hold at that point. Formally (the law is originally due to Singh [11]):

Theorem 2.7 : (Generalized) Singh Law

$$
\frac{\begin{array}{l} P, J \vdash p \; Rel \; q \text{ using } V \qquad a, b \textbf{ conf } W \\ Q, J \vdash V \text{ is preserved when } a \text{ unless } b \end{array}}{P\|Q, J \vdash (p \wedge a) \; Rel \; ((q \wedge a) \vee \neg a \vee b) \text{ using } V \cup W}
$$

Below are some useful and powerful corollaries of the Singh Law.

Theorem 2.8 : Until Composition

$$
\frac{P \vdash p \text{ until } q \text{ using } V \qquad Q \vdash V \text{ is preserved when } p \text{ unless } q}{P\|Q \vdash p \text{ until } q \text{ using } V}
$$

The problem with the above theorem is that it requires P to maintain the 'whole' pre-condition (p) of the progress at least until q holds. In some cases this is too stringent. A more realistic scenario is that, when p holds, P sets a 'flag' a to indicate it wants to use the resources listed in V and maintains a until it reaches q. Then, while a is raised, program Q suspends its interference on V at least until q holds, then $a \wedge p \mapsto q$ using V can be preserved in $P\|Q$. Notice that in this scenario P only needs to maintain a (rather than p). This is, slightly generalized, formalized by the Scheduling Law below:

Theorem 2.9 : Scheduling

$$
\frac{\begin{array}{l} P \vdash p \mapsto q \text{ using } V \quad , \quad P \vdash a \text{ unless } q \text{ using } V \\ Q \vdash V \text{ is preserved when } a \text{ unless } r \end{array}}{P\|Q \vdash a \wedge p \; Rel \; q \vee r \text{ using } V}
$$

Many types of synchronization are instances of the Scheduling Law. An example will be shown later in Section 3.

3 Example

Consider a system Sys consisting of N components $P_1 \ldots P_N$ that share common resources: display and a 'clipboard' c. The *clipboard* can be used to pass data from one component to another, or temporarily hold internal data of a component. The clipboard is modeled by a list: [] denotes an empty clipboard, and $i{:}s$ means that it contains the data s belonging to component P_i. When the clipboard is empty, any component can fill it with a request X for component P_i to update the display to $\phi(i{:}X)$. This is done by putting $i{:}X$ in the clipboard. P_i only accepts a request if it is idle, a state that will be indicated by a boolean variable idle $_i$. During the processing of a request P_i is allowed to use c to store intermediate results. Once the display is updated, the clipboard should be emptied again so that other components can use it. Consider the following informal specification of each P_i: upon a request $c = i{:}X$, when P_i is idle, eventually it will set the display to the correct value and empties the clipboard c. During this time, we expect P_i to require at least c, display and idle $_i$ as resources. Using our operators, this can be formally specified by:

$$\mathsf{Sys}, J \;\vdash\; \begin{aligned} &(c = i{:}X) \wedge \text{ idle }_i \;\mapsto\; (\text{display} = \phi(i{:}X)) \wedge (c = [\,]) \\ &\text{using} \\ &\{c, \text{display, idle }_i\} \cup V \end{aligned} \tag{2}$$

for some 'invariant' J and some set of additional resources V. The specification is expressed as a global property of Sys. Our task is now to obtain the specification of P_i and a constraint for the other components such that when combined they imply the above property.

The SINGH LAW (Theorem 2.7) can be used to derive the properties that the environment should satisfy in order to preserve P_i's progress from (2) in the composition $P_1 [\![P_2 [\![\ldots [\![P_N$. However, because of its generality, the law itself offers little hint for us in figuring out what constraints we can reasonably impose on the other components P_j's without overly constraining P_i.

The corollaries offer more. For example, the UNTIL COMPOSITION LAW (2.8) says that if we strengthen the $\mapsto$ property in (2) to an until property, it can be preserved when the other components P_j satisfy a matching unless. Unfortunately, requiring P_i to maintain the condition $c = i{:}X$ until P_i has correctly set up the display is too stringent because earlier we have indicated that P_i could use the clipboard c to temporarily store its intermediate data.

Let us now try the SCHEDULING LAW (Theorem 2.9). First we have to decide which "flag" – i.e. predicate a in Theorem 2.9– we are going to use. It is a condition that, if it is entered by the component P_i, will trigger its environment to suspend interference on the resources claimed by its specification in (2). Note that whatever condition we choose as the flag, the SCHEDULING LAW law requires the flag to remain raised until the objective of the progress in (2) is achieved. For example, idle $_i$ is not a good candidate for the flag, because, although we did not mention it, it is reasonable to assume that P_i will flip idle $_i$ to false as soon as it starts processing the clipboard. The condition $c = i{:}X$ cannot be the flag either, because, as indicated before, P_i may use the second component of c to store its intermediate data. However, since it is acceptable

to expect that P_i will not change the ownership indicator (the i part of c), we can assume the first component of c to remain the same during the progress in (2). That is, $c \neq [] \ \wedge \ \mathsf{hd} \ c = i$ (let f abbreviate this expression) can be expected to hold during the progress in (2), while still allows P_i to change the data contents of the clipboard (the X part). Consequently, f is a good flag candidate. However, there is still a small problem. As indicated before, the Scheduling Law requires the flag to remain raised at least until the display is correctly set, thus until $\mathsf{display} = \phi(i{:}X)$ where X is the original value of the data content of the clipboard. Unfortunately, we no longer have the value X in f, and thus cannot test if it still the same. To cope with this, we introduce an auxiliary variable d_i to hold the original data contents X of c, until the objective of the progress specified in (2) is reached. This variable can be just a 'virtual' variable, in the sense that it does not really have to be implemented in an actual P_i. We just need it to be able to express some relation between old and new values in our properties[1]. Using this new variable d_i, we decide that the flag we are going to use is:

$$(c \neq []) \wedge (\mathsf{hd} \ c = i) \wedge (d_i = X) \tag{3}$$

Consider now the following specification, which if it can be preserved in Sys implies (2):

$$P_i, J \vdash \quad (c = i{:}X) \wedge \ \mathsf{idle} \ _i \ \mapsto \ (\mathsf{display} = \phi(i{:}X)) \wedge (c = [])$$
$$\mathsf{using}$$
$$\{c, \mathsf{display}, \ \mathsf{idle} \ _i, d_i\} \tag{4}$$

Now, in order to preserve this progress in Sys, the Scheduling Law (where a is (3) in Theorem 2.9) sets two conditions. One is an unless condition set on P_i:

$$P_i, J \vdash \quad (c \neq []) \wedge (\mathsf{hd} \ c = i) \wedge (d_i = X) \ \ \mathsf{unless} \ \ (\mathsf{display} = \phi(i{:}X)) \wedge (c = [])$$
$$\mathsf{using}$$
$$\{c, \mathsf{display}, \ \mathsf{idle} \ _i, d_i\} \tag{5}$$

and the other is a preservation condition on all P_j's for which j distinct from i:

$$P_j, J \vdash \quad \{c, \mathsf{display}, \ \mathsf{idle} \ _i, d_i\} \ \text{is preserved when} \ (c \neq []) \wedge (\mathsf{hd} \ c = i) \wedge (d_i = X)$$
$$\mathsf{unless}$$
$$(\mathsf{display} = \phi(i{:}X)) \wedge (c = []) \tag{6}$$

It is actually reasonable to require something stronger from P_j, namely that it does not interfere with any common resource while P_i is still busy –the state of the latter can be read from the state of the flag chosen earlier. Since we can weaken the right argument of unless (remarked earlier), we can instead impose the following:

$$P_j, J \vdash \quad \{c, \mathsf{display}, \ \mathsf{idle} \ _i, d_i\} \ \text{is preserved when} \ (c \neq []) \wedge (\mathsf{hd} \ c = i) \wedge (d_i = X)$$
$$\mathsf{unless} \ \ false \tag{7}$$

---

[1] We have restricted our temporal operators to be first order. One of the consequences is that we sometimes have to resort to using virtual variables to record and refer to old program states.

This requirement is stronger than (6), but simpler. If we insists that d_i is a local variable of P_i (which is a reasonable assumption), then using Theorem 2.2 and Theorem 2.3 we can simplify (7) even further by dropping the conjunct $d_i = X$ and variable d_i. So we obtain this:

$$P_j, J \vdash \quad \{c, \mathsf{display}, \ \mathsf{idle}_i\} \text{ is preserved when } (c \neq []) \land (\mathsf{hd} \ c = i) \\ \mathsf{unless} \ \ \mathsf{false} \tag{8}$$

If idle_i is also a local variable of P_i (which seems reasonable to assume), then Theorem 2.2 also says that we can drop idle_i from the confining set in the above specification, and specify the requirements on the environment finally by:

$$P_j, J \vdash \{c, \mathsf{display}\} \text{ is preserved when } (c \neq []) \land (\mathsf{hd} \ c = i) \text{ unless false} \tag{9}$$

To summarize, above we have derived a constraint on P_i, namely (5), and a constraint on its environment, namely (9), which are sufficient to ensure that (4) is preserved in the composed system $P_1 [\![P_2 [\![\ldots [\![P_N]$, and thus implying the original specification (2).

4 Conclusion

When reasoning about distributed components, it is important to be able to specify when a given set of common resources is needed by a component, and when they will be released again such that other components can use them. We call this temporary interference. We have extended some standard temporal operators to capture temporary interference and proved a set of laws to reason about those extended operators. By exploiting information about temporary interference we have constructed a nice and intuitive theory to prove that specific temporal properties of a component can be preserved in a parallel composition. It is not a general theory, as it requires that the components take turn in writing to common resources, but we believe that it still covers a large class of useful systems. Moreover, in exchange for its incompleteness, the verification constraints generated in order to match a component with its environment are relatively easy to verify. The soundness of the theory with respect to a UNITY-like semantics has been mechanically verified in HOL (a theorem prover [9]) –see [10] principle the theory can be made more general though of course we will then have to give up some of its strength —see [10] for more about this issue.

We currently have no serious case study to see how effective the theory actually is in practice, so this is future work.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

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

[3] K. Chandy and M. Charpentier. An experiment in program composition and proof. *Formal Methods in System Design*, 20(1):7–21, 2002.

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

[5] K.M. Chandy and B.A. Sanders. Reasoning about program composition. Technical Report 96-035, University of Florida, 1996.

[6] M. Charpentier and K. Chandy. Theorems about composition. *Lecture Notes of Computer Science*, 1837:167–186, 2000.

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

[8] P. Collette and E. Knapp. Logical foundations for compositional verification and development of concurrent programs in UNITY. *Lecture Notes of Computer Science*, 936:353 – 367, 1995.

[9] Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[10] I.S.W.B. Prasetya, T.E.J. Vos, S.D. Swierstra, and B. Widjaja. A theory for composing distributed components based on mutual exclusion, 2002. Draft. Download: `www.cs.uu.nl/~wishnu`.

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