ELSEVIER

Theoretical Computer Science 243 (2000) 339–361

Theoretical Computer Science

www.elsevier.com/locate/tcs

Composing leads-to properties

David Meier [a], Beverly Sanders [b,*]

[a] *Pilgerweg 1, 8044 Zurich, Switzerland*

[b] *Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611-6120 USA*

Received April 1996; revised June 1998

Communicated by M. Sintzoff

Abstract

Compositionality is of great practical importance when building systems from individual components. Unfortunately, leads-to properties are not, in general, compositional, and theorems describing the special cases where they are, are needed. In this paper, we develop a general theory of compositional leads-to properties, and use it to derive a composition theorem based on the notion of progress sets, where progress sets can be defined in various ways. Appropriate definitions of progress sets yield new results and generalized versions of known theorems.

Keywords: parallel composition; verification; leads-to; commutativity

1. Introduction

Although leads-to properties are not compositional, in general, it is worthwhile to identify the special cases where they are. Composition theorems for leads-to properties have been proposed, for example, in [15, 16, 19]. In this paper, we develop a general theory about composition of leads-to properties, then specialize the results to give a composition theorem based on the notion of progress sets. A progress set for a program F and target q is a set of predicates closed under taking conjunctions and disjunctions and satisfying certain properties expressible by weakest preconditions. The theorem essentially states that if for each predicate in the progress set, program G satisfies a particular property, then any "leads-to q" property that holds for F, also holds in the parallel composition of F and G. Several different composition theorems can be obtained by choosing particular ways of constructing progress sets.

* Corresponding author.

E-mail address: sanders@cise.ufl.edu (B. Sanders).

PII: S0304-3975(98)00233-3

First, we will introduce our program model, define the necessary background information, and develop a very general theorem for composing programs in a way that preserves leads-to properties. This theorem is then specialized to obtain the main result of the paper – a composition theorem based on progress sets. Finally, we explore different choices for the way a progress set is constructed and give several useful corollaries.

2. Preliminaries

2.1. Programs and properties

A program F is a pair (V_F, S_F) where V is a set of typed variables and S is a set of predicate transformers that includes the identity transformer and represents the weakest preconditions of a set of nonmiraculous, always terminating, and boundedly nondeterministic commands. Thus each $s \in S_F$ is universally conjunctive, strict w.r.t. *false*, and or-continuous. Since the identity transformer corresponds to the command *skip*, all programs in our model allow stuttering steps. The state space of F is a Cartesian product with a coordinate for each variable of V. If V_F is the empty set, the state space is a single state representing the empty Cartesian product.

A computation of F is an initial state σ_0, and a sequence of pairs (s_i, σ_i), $i > 0$, where $s_i \in S$ is a command and σ_i is a program state, and execution of command s_i can take the program from state σ_{i-1} to state σ_i, for $i > 0$, and each command in the program appears infinitely often. This definition follows the one in [4] or may also be viewed as a generalized version of UNITY [3] with no initially section. In contrast with UNITY, our model does not explicitly restrict the initial condition of a program. Thus we will not be able to use a rule such as the UNITY substitution axiom to eliminate states unreachable from a specified initial state from consideration. We will, however, take into account that after *any* point in the computation, not just the initial one, some states may be no longer reachable or are not reachable in some interval of interest.

Now we define several predicate transformers and program properties. The square brackets in the definitions are the everywhere operator from [7].

The predicate transformers *awp.F* **and properties** *co* **and** *stable***.** The predicate transformer *awp* [4], is defined as

$$[awp.F.q \equiv (\forall s : s \in S_F : s.q)]. \tag{1}$$

Using *awp*.F, we define a property *co.F* [4, 17] that describes the next state relation of the program F.

$$p \ co_F \ q = [p \Rightarrow q \wedge awp.F.q]. \tag{2}$$

Operationally, $p \ co_F \ q$ means that if p holds at some point during a computation, then q holds and will still hold after executing any command of F.

The property *stable.F.q* is defined as

$$stable.F.q = [q \Rightarrow awp.q], \tag{3}$$

which indicates that q will never be falsified by any command of F.

Refinement of *awp.F*. For two programs F and F', we say that F is refined by F',[1] denoted $F \leqslant F'$ when the following formula holds.

$$(\forall p : [awp.F.p \Rightarrow awp.F'.p]) \tag{4}$$

For our purposes in this paper, we use the fact that if F is refined by F', every *co* property of F is also a *co* property of F'.

The predicate transformer *wens.F.s*, **and properties** *ensures* **and leads-to (**$\rightsquigarrow$**).** The weakest predicate that will hold until q does, and which will be taken to q by a single s step is denoted *wens.F.s.q*. It is defined as

$$[wens.F.s.q \equiv \nu x : [x \equiv (s.q \wedge awp.F.(q \vee x)) \vee q]]. \tag{5}$$

The predicate transformer *wens.F.s* is monotonic (i.e. $[p \Rightarrow q] \Rightarrow [wens.F.s.p \Rightarrow wens.F.s.q]$) and weakening (i.e. $[q \Rightarrow wens.F.s.q]$).

From predicate calculus and the fixed point induction rule,

$$[p \wedge \neg q \Rightarrow (s.q \wedge awp.F.(q \vee p))] \Rightarrow [p \Rightarrow wens.F.s.q]. \tag{6}$$

From [4] we have a rule stating that if a predicate is stable, then its *wens* is also stable.

$$stable.F.q \Rightarrow stable.F.(wens.F.s.q). \tag{7}$$

If $[t \wedge \neg q \Rightarrow awp.F.t]$ holds, then operationally, we have that if $t \wedge \neg q$ holds, at some point, then t will continue to hold while $\neg q$ does, and it will also hold after the step that has established q. In this case, the states that satisfy both t and *wens.F.s.q* are the same as those satisfying both t and $wens.F.s.(q \wedge t)$:

$$[t \wedge \neg q \Rightarrow awp.F.t] \Rightarrow [t \wedge wens.F.s.q \equiv t \wedge wens.F.s.(t \wedge q)]. \tag{8}$$

Proof of (8). The proof is by mutual induction.

$$\begin{array}{ll} & true \\ = & \quad \{\text{With } [p \equiv wens.F.s.q], \text{ from the definition of } wens\ (5)\} \\ & [p \equiv s.q \wedge awp.F.(p \vee q) \vee q] \\ \Rightarrow & \quad \{\text{predicate calculus}\} \\ & [p \wedge \neg q \Rightarrow s.q \wedge awp.F.(p \vee q)] \\ \Rightarrow & \quad \{\text{hypothesis, } [t \wedge \neg q \Rightarrow awp.F.t]\} \end{array}$$

[1] This definition of refinement only considers safety properties. A more general notion, where progress properties are taken into account is described in [21].

$$
\begin{array}{ll}
 & [p \wedge t \wedge \neg q \Rightarrow s.q \wedge awp.F.(p \vee q) \wedge awp.F.t] \\
= & \quad \{ \text{ def } awp, \wedge \text{ idempotent, thus } [awp.F.t \equiv awp.F.t \wedge s.t]\} \\
 & [p \wedge t \wedge \neg q \Rightarrow s.q \wedge s.t \wedge awp.F.(p \vee q) \wedge awp.F.t] \\
= & \quad \{s, \ awp.F \text{ conjunctive}\} \\
 & [p \wedge t \wedge \neg q \Rightarrow s.(q \wedge t) \wedge awp.F.((p \vee q) \wedge t)] \\
= & \quad \{\text{predicate calculus}\} \\
 & [(p \wedge t) \wedge \neg(q \wedge t) \Rightarrow s.(q \wedge t) \wedge awp.F.((p \wedge t) \vee (q \wedge t))] \\
\Rightarrow & \quad \{ \ (6), \ p := (p \wedge t), \ q := (q \wedge t)\} \\
 & [(p \wedge t) \Rightarrow wens.F.s.(q \wedge t)] \\
= & \quad \{\text{predicate calculus, p} := \text{wens.F.s.q}\} \\
 & [(t \wedge wens.F.s.q) \Rightarrow t \wedge wens.F.s.(q \wedge t)]
\end{array}
$$

and

$$
\begin{array}{ll}
 & true \\
= & \quad \{\text{predicate calculus}\} \\
 & [(q \wedge t) \Rightarrow q] \\
\Rightarrow & \quad \{wens.F.s \text{ monotonic}\} \\
 & [wens.F.s.(q \wedge t) \Rightarrow wens.F.s.q] \\
\Rightarrow & \quad \{\wedge \text{ monotonic}\} \\
 & [t \wedge wens.F.s.(q \wedge t) \Rightarrow t \wedge wens.F.s.q].
\end{array}
$$

The ensures property $p \ ensures \ q$ in F [3] can be defined as

$$p \ ensures \ q \equiv (\exists s : [p \Rightarrow wens.F.s.q]). \tag{9}$$

Operationally, this means that if at some point $p \wedge \neg q$ holds, then eventually q will hold, and furthermore, there is a single command that, when executed, will establish q.

From (8), we easily obtain

$$t \wedge \neg q \ co_F \ t \wedge p \ ensures \ q \Rightarrow p \wedge t \ ensures \ q \wedge t. \tag{10}$$

From [3], $p \leadsto_F q$, read p leads-to q in F, is the unique strongest relation (on predicates) satisfying

$$[p \ ensures \ q] \Rightarrow p \leadsto_F q, \tag{11}$$

$$[p \ ensures \ r] \wedge r \leadsto_F q \Rightarrow p \leadsto_F q, \tag{12}$$

$$(\forall w : w \in W : w \leadsto_F q) \Rightarrow (\exists w : w \in W : w) \leadsto_F q, \tag{13}$$

where W is an arbitrary set of predicates on the state space of F.

A function from programs to predicate transformers *wlt* (weakest leads-to) has been given in [10, 11] where

$$p \leadsto_F q = [p \Rightarrow wlt.F.q] \tag{14}$$

We will give a new formulation of *wlt* below.

Notation for theorems. We often write theorems in the following form: First we list the preconditions, each beginning on a new line, then follows the implication arrow or an equivalence sign, after it the conclusion of the theorem.

2.2. The weakest ensures set of q, $E.F.q$

Now, we introduce a new concept. For program F and predicate q, $E.F.q$ is defined as the minimal set of predicates satisfying

$$q \in E.F.q, \tag{15}$$

$$p \in E.F.q \Rightarrow wens.F.s.p \in E.F.q, \tag{16}$$

$$(\forall w : w \in W : w \in E.F.q) \Rightarrow (\exists w : w \in W : w) \in E.F.q, \tag{17}$$

where W is an arbitrary set of predicates on the state space of F.

Note that the two properties (15) and (16) are closely related to the base property (11) and the transitivity property (12) of *leads – to* above, and that (17) corresponds to (13) of the *leads – to* definition.

Some properties of $E.F.q$ that will be used later are

$$p \in E.F.q \Rightarrow [q \Rightarrow p] \tag{18}$$

and

$$p \leadsto_F q = (\exists r : r \in E.F.q : [p \Rightarrow r]) \tag{19}$$

or alternatively,

$$wlt.F.q \equiv (\exists r : r \in E.F.q : r). \tag{20}$$

Induction on the structure of $E.F.q$. We will frequently need to show that all predicates in the weakest ensures set $E.F.q$, have a certain property, for example that all are $stable.G$ for some program G. A look at the three conditions (15), (16) and (17) shows that we can prove that all elements of $E.F.q$ have a property by showing that

(1) the property holds for q,
(2) if the property holds for r then for all s, the property holds for $wens.F.s.r$,
(3) if the property holds for all r_i with $i \in I$ then the property holds for $(\exists i : i \in I : r_i)$.

Several proofs in the sequel will use induction on the structure of $E.F.q$.

2.3. Parallel Composition

For two programs $F = (V_F, S_F)$ and $G = (V_G, S_G)$, their parallel composition or union, denoted $F \| G$ is defined as

$$F \| G = (V_F \cup V_G, S_F \cup S_G). \tag{21}$$

Parallel composition is only defined when common elements of V_F and V_G have the same type, however, we will assume that $F\|G$ is defined whenever we write it. Predicates on the state space of, say F, may be also be viewed as predicates on the state space of $F\|G$ since the union may only increase the number of variables.

The following theorems follow easily from the definitions:

$$[awp.F.q \wedge awp.G.q \equiv awp.F\|G.q], \tag{22}$$

$$[wens.F\|G.s.q \Rightarrow wens.F.s.q], \tag{23}$$

$$\begin{aligned}&([p = t \wedge wens.F.s.q] \wedge [\neg q \wedge t \Rightarrow awp.F.t] \wedge\\ &[p \wedge \neg q \Rightarrow awpG(p \vee q)])\\ &\Rightarrow\\ &[p \Rightarrow wensF\|Gsq]\end{aligned} \tag{24}$$

The asymmetry of (23) and (24) is due to the asymmetry of the problem we study. We assume that we know that $p \leadsto_F q$ and look for conditions for which this implies $p \leadsto_{F\|G} q$.

3. Composing leads-to properties

While *co* and *ensures* admit simple composition theorems [3], simple composition theorems do not hold, in general, for leads-to properties. We can, however, use the relationship between leads-to properties and the elements of $E.F.q$ to give a general composition theorem that provides a starting point for more useful theorems.

Below, we give a union theorem for $E.F.q$. Intuitively, the theorem says that if for every predicate r in $E.F.q$, if $r \wedge \neg q$ holds at some point, then r continues to hold until q is established, then r is also in $E.F\|G.q$. The theorem is actually more general, introducing a predicate t satisfying $[t \wedge \neg q \Rightarrow awp.F.t]$. This allows us, in essence, to restrict attention to the parts of the state space satisfying t.

Union Theorem for $E.F.q$.

$$[\neg q \wedge t \Rightarrow awp.F.t] \tag{25}$$

$$(\forall r : r \in E.F.q : [\neg q \wedge (r \wedge t) \Rightarrow awpG(r \wedge t)]) \tag{26}$$

$\Rightarrow$

$$(\forall r : r \in E.F.q : (\exists r' : r' \in E.F\|G.q : [r \wedge t \equiv r' \wedge t])). \tag{27}$$

Proof of Union Theorem for $E.F.q$**.** We show a stronger result:

$$(\forall r : r \in E.F.q : (\exists r' : r' \in E.F\|G.q : [r' \Rightarrow r] \wedge [r \wedge t \equiv r' \wedge t])).$$

The proof is by induction on the structure of the set $E.F.q$.

Base:

From (15), for $r = q$ we have $r' = q$.

Induction with (16):

Let $r = wens.F.s.x$ and $r' = wens.F\|G.s.x'$.

The induction hypothesis is $[x \wedge t \equiv x' \wedge t] \wedge [x' \Rightarrow x]$, $x \in E.F.q$

$*$ $wens.F.s.x \wedge t$

$\equiv$ $\{(8),(25)\}$

$wens.F.s.(x \wedge t) \wedge t$

$\Rightarrow$ $\{(24)$ with

$p := wens.F.s.(x \wedge t) \wedge t$,

$q := t \wedge x$,

and using that $p \wedge \neg q \Rightarrow awp.G.(p \vee q)$ follows

from (26) and $[q \Rightarrow x]\}$

$wens.F\|G.s.(x \wedge t)$

$\equiv$ $\{$induction hypothesis, $[x \wedge t \equiv x' \wedge t]\}$

$wens.F\|G.s.(x' \wedge t)$

$\Rightarrow$ $\{wens.F\|G.s$ monotonic$\}$

$**$ $wens.F\|G.s.(x')$

$\Rightarrow$ $\{wens.F\|G.s$ monotonic, induction hypothesis, $[x' \Rightarrow x]\}$

$wens.F\|G.s.(x)$

$\Rightarrow$ $\{(23)\}$

$***$ $wens.F.s.(x)$.

Thus we have, from the starred lines in the proof that

$[r' \Rightarrow r]$ and $[r \wedge t \equiv r' \wedge t]$.

Induction with (17):

Follows from the predicate calculus.

As a simple consequence of the union theorem for $E.F.q$ and (19), we obtain a union theorem for leads-to.

Leads-to Union Theorem

$$[\neg q \wedge t \Rightarrow awp.F.t], \tag{28}$$

$$(\forall r : r \in E.F.q : [\neg q \wedge (r \wedge t) \Rightarrow awp.G.(r \wedge t)]), \tag{29}$$

$\Rightarrow$

$$(wlt.F.q) \wedge t \rightsquigarrow_{F\|G} q. \tag{30}$$

Proof. From the union theorem for $E.F.q$ and the fact that $wlt.F.q \in E.F.q$, we have $(\exists r' : r' \in E.F\|G.q : [wlt.F.q \wedge t \equiv r' \wedge t])$. Thus $[(wlt.F.q) \wedge t \Rightarrow r']$, which, together with (19), imply $(wlt.F.q) \wedge t \rightsquigarrow_{F\|G} q$.

A corollary of the above is the following.

Corollary to the leads-to union theorem

$$
\begin{array}{l}
\quad p \leadsto_F q \\
\quad [\neg q \wedge t \Rightarrow awp.F.t] \\
\quad (\forall r : r \in E.F.q : [\neg q \wedge (r \wedge t) \Rightarrow awp.G.(r \wedge t)]) \\
\Rightarrow \\
\quad p \wedge t \leadsto_{F\|G} q.
\end{array}
$$

Note that if t is such that $[p \Rightarrow t]$, then the conclusion is $p \leadsto_{F\|G} q$.

Example. In the next example we explore the composition of two simple single statement programs. The variable x is an integer.

$$F : x := x + 1,$$

$$G : x := 2x.$$

Let $q = (x \geqslant k)$, for some k. First, we determine $E.F.q$. For a program with a single command s, $wens.F.s.q = q \vee s.q$. Using now $s.(x \geqslant i) = (x + 1 \geqslant i)$, we get

$$E.F.q = \{i : i \leqslant k : (x \geqslant i)\} \cup \{true\}$$

and $[wlt.F.(x \geqslant k) \equiv true]$. The union theorem can be applied, provided

$$(\forall r : r \in E.F.q : [\neg(x \geqslant k) \wedge r \wedge t \Rightarrow (awp.G.(r \wedge t)]), \tag{31}$$

where $[awp.G.(r \wedge t) \equiv (x := 2x)(r \wedge t)]$. It is easy to see that (31) does not hold for any k if t is *true*, however, it does hold for all k if t is $x \geqslant 0$. In addition, this choice of t satisfies (28). Thus we can conclude from the analysis that

$$x \geqslant 0 \leadsto_{F\|G} x \geqslant k.$$

The example indicates the importance of being able to restrict the state space under consideration. The condition on t essentially says that once t holds in $F\|G$ then it will continue to hold at least until q, the target, does. We use this to weaken the requirements on G at the price of having t on the left side of the leads-to properties of the composed program.

In most cases, one is concerned that a particular progress property of F, say $(x \geqslant 3) \leadsto (x \geqslant 10)$ is preserved in a composition, and the corollary is applicable. In this case $(x \geqslant 3) \wedge (x \geqslant 0)$ is just $(x \geqslant 3)$. Even though the restriction of the state space to states satisfying $(x \geqslant 0)$ has not impacted the conclusion, it was still needed to apply the theorem.

The above example shows the usefulness of the predicate t. Because the theorems are considerably simpler with $[t \equiv true]$, we rewrite two of them below.

Union Theorem for $E.F.q$ **for** $[t \equiv true]$

$$(\forall r : r \in E.F.q : [\neg q \wedge r \Rightarrow awp.G.r])$$
$$\Rightarrow$$
$$E.F.q \subseteq E.F \| G.q.$$

Corollary to the leads-to union theorem for $[t \equiv true]$

$$p \leadsto_F q$$
$$(\forall r : r \in E.F.q : [\neg q \wedge r \Rightarrow awp.G.r])$$
$$\Rightarrow$$
$$p \leadsto_{F \| G} q.$$

A generalization of Misra's fixed point union theorem Directly applying the union theorem for leads-to is usually not practical. However, we show below how it can be specialized to yield Misra's fixed point union theorem.

A predicate q is a fixed point if the state no longer changes once q holds. Formally,

$$q \text{ is a fixed point of } G = (\forall p : stable.G.(p \wedge q)).$$

Once a program has reached a fixed point, its state no longer changes. The following theorem follows is a simple corollary of the leads-to union theorem since, from the definition of fixed point, (29) holds for all predicates.

Generalized fixed point union theorem

$$p \leadsto_F q$$
$$[\neg q \wedge t \Rightarrow awp.F.t]$$
$$\neg q \wedge t \text{ is a fixed point of } G \tag{32}$$
$$\Rightarrow$$
$$p \wedge t \leadsto_{F \| G} q.$$

Misra's version [15] is obtained from the above with $[t \equiv true]$ and the observation that in this case $[\neg q \wedge t \Rightarrow awp.F.t]$ hold trivially.

4. A composition theorem using progress sets

In this section, we give conditions under which a set $C.F.q$ of predicates is guaranteed to contain $E.F.q$. The idea is that this set of predicates is easier to describe than $E.F.q$.

First, we require that $C.F.q$ is closed under arbitrary conjunction and disjunction. For arbitrary R:

$$(\forall r : r \in R : r \in C) \Rightarrow (\forall r : r \in R : r) \in C, \tag{33}$$

$$(\forall r : r \in R : r \in C) \Rightarrow (\exists r : r \in R : r) \in C. \tag{34}$$

Since R may be empty, the above formulae imply that $true \in C$ and $false \in C$.

The predicate transformer $cl.C$**.** Given the set C, we define a predicate transformer $cl.C$, where $cl.C.r$ is the strongest predicate in C weaker than r:

$$cl.C.r \equiv \mu x : x \in C \wedge [r \Rightarrow x]. \tag{35}$$

The postulated properties of C are sufficient to guarantee the existence of $cl.C.r$. In addition, $cl.C.r$ is monotonic, weakening, and universally disjunctive. Also,

$$[cl.C.r \equiv r] = r \in C. \tag{36}$$

Progress sets. For a program F, and predicates t and q, we say that C is a progress set for the triple (F, t, q) if

$$C \text{ is closed under arbitrary conjunction and disjunction } (33, 34) \tag{37}$$

$$q \in C, \tag{38}$$

$$t \in C \wedge stable.F.t, \tag{39}$$

$$(\forall m, s : [q \Rightarrow m], s \in F : (t \wedge m) \in C \Rightarrow (t \wedge (q \vee s.m)) \in C). \tag{40}$$

As in the previous section the predicate t allows us to restrict attention to the parts of the state space satisfying t. Conditions (38) and (40) say that q belongs to the progress set for (F, t, q) and that, in a generalized form, the progress set is closed under taking weakest preconditions in F.

The following lemma says that if C is a progress set for (F, t, q), then for all predicates $r \in E.F.q$, $(r \wedge t) \in C$. The lemma will allow us to reformulate the leads-to union theorem in terms of a progress set instead of $E.F.q$.

Lemma.

$$\begin{aligned} & C \text{ is closed under arbitrary conjunction and disjunction} \\ & q \in C \\ & t \in C \wedge stable.F.t \\ & (\forall m, s : [q \Rightarrow m], s \in S_F : (t \wedge m) \in C \Rightarrow (t \wedge (q \vee s.m)) \in C) \\ \Rightarrow & \\ & (\forall r : r \in E.F.q : (r \wedge t) \in C). \end{aligned}$$

The proof is by induction on the structure of $E.F.q$. The base case follows from the hypothesis (i.e. $q \in C, t \in C$) and the fact that C is closed under conjunction (33). The induction step with disjunction follows from the fact that C is closed under disjunction (34). The remaining induction requires us to show that:

$$(t \wedge m) \in C \Rightarrow (t \wedge wens.F.s.m) \in C. \tag{41}$$

Let $r = wens.F.s.m$,

$$
\begin{array}{ll}
 & true \\
= & \quad \{\text{definition of } wens.s,\ (5)\} \\
 & [r \Rightarrow (s.m \wedge awp.F.(m \vee r)) \vee m] \\
\Rightarrow & \quad \{\text{predicate calculus}\} \\
 & [r \wedge t \Rightarrow t \wedge ((s.m \wedge awp.F.(m \vee r)) \vee m)] \\
= & \quad \{stable.F.t \text{ thus } [t \equiv awp.t]\} \\
 & [t \wedge r \Rightarrow t \wedge ((s.m \wedge awp.F.(m \vee r) \wedge awp.F.t) \vee m)] \\
= & \quad \{awp.F. \text{ conjunctive}\} \\
 & [t \wedge r \Rightarrow t \wedge (s.m \wedge awp.F.((m \vee r) \wedge t) \vee m)] \\
\Rightarrow & \quad \{awp \text{ monotonic, } [((m \vee r) \wedge t) \Rightarrow (m \vee (r \wedge t)], \text{ weaken right side}\} \\
 & [t \wedge r \Rightarrow t \wedge (s.m \wedge awp.F.(m \vee (r \wedge t)) \vee m)] \\
\Rightarrow & \quad \{cl..C. \text{ weakening, } awp.F. \text{ monotonic, weaken right side}\} \\
 & [t \wedge r \Rightarrow t \wedge (s.m \wedge awp.F.(m \vee cl.C.(r \wedge t)) \vee m)] \\
\Rightarrow & \quad \{cl.C. \text{ monotonic}\} \\
 & [cl.C.(t \wedge r) \Rightarrow cl.C.(t \wedge (s.m \wedge awp.F.(m \vee cl.C.(r \wedge t)) \vee m))] \\
= & \quad \{(t \wedge (s.m \wedge awp.F.(m \vee cl.C.(r \wedge t)) \vee m)) \in C, \text{ see below}\} \\
 & [cl.C.(t \wedge r) \Rightarrow t \wedge (s.m \wedge awp.F.(m \vee cl.C.(r \wedge t)) \vee m)] \\
\Rightarrow & \quad \{\text{weaken right side}\} \\
 & [cl.C.(t \wedge r) \Rightarrow s.m \wedge awp.F.(m \vee cl.C.(r \wedge t)) \vee m] \\
\Rightarrow & \quad \{\text{fixed point, } (6)\} \\
 & [cl.C.(t \wedge r) \Rightarrow wens.F.s.m] \\
= & \quad \{\text{definition of } r\} \\
 & [cl.C.(t \wedge r) \Rightarrow r] \\
\Rightarrow & \quad [t \equiv cl.C.t],\ cl.C. \text{ monotonic, thus } [cl.C.(t \wedge r \Rightarrow t] \\
 & [cl.C.(t \wedge r) \Rightarrow t \wedge r] \\
\Rightarrow & \quad \{cl.C. \text{ weakening, thus } [t \wedge r \Rightarrow cl.C.(t \wedge r)]\} \\
 & [t \wedge r \equiv cl.C.(t \wedge r)] \\
= & \quad \{\text{definition of } r, \text{ property of } cl.C., \ (36)\} \\
 & t \wedge wens.F.s.m \in C.
\end{array}
$$

Now we show

$$(t \wedge (s.m \wedge awp.F.(m \vee cl.C.(r \wedge t)) \vee m)) \in C$$

that was assumed above. By the induction hypotheses $t \wedge m \in C$.

$$
\begin{array}{ll}
 & true \\
= & \quad \{\text{induction hypothesis}\} \\
 & (t \wedge m) \in C \\
\Rightarrow & \quad \{C \text{ closed under disjunction}\} \\
 & ((t \wedge m) \vee cl.C.(r \wedge t)) \in C \\
\Rightarrow & \quad \{\text{predicate calculus, using } [cl.C.(r \wedge t) \Rightarrow t]\} \\
 & (t \wedge (m \vee cl.C.(r \wedge t)) \\
\Rightarrow & \quad \{\text{hypothesis}\} \\
 & (t \wedge (q \vee s.(m \vee cl.C.(r \wedge t)))) \in C
\end{array}
$$

$$
\begin{array}{ll}
\Rightarrow & \{\text{conjunction over } s \in T_F,\ C \text{ closed under conjunction}\} \\
& (t \wedge (q \vee awp.F.(m \vee cl.C.(r \wedge t)))) \in C \\
\Rightarrow & \{\text{induction hypothesis and hypothesis of lemma give}\} \\
& (t \wedge (q \vee s.m)),\ C \text{ closed under conjunction}\} \\
& (t \wedge (q \vee (s.m \wedge awp.F.(m \vee cl.C.(r \wedge t))))) \in C \\
\Rightarrow & \{(t \wedge m),\ C \text{ closed under disjunction, } [q \Rightarrow m]\} \\
& (t \wedge (m \vee (s.m \wedge awp.F.(m \vee cl.C.(r \wedge t))))) \in C).
\end{array}
$$

A monotonicity property of progress sets. From the conjunction and disjunction properties of a progress set C for (F,t,q) one derives the following monotonicity property:

$$
\begin{array}{l}
\quad C \text{ is a progress set for } (F,t,q) \\
\quad [q \Rightarrow q'] \wedge q' \in C \\
\Rightarrow \\
\quad C \text{ is a progress set for } (F,t,q').
\end{array}
$$

Now, we have the final theorem of this section, and a main result of the paper. It follows immediately from the lemma and the leads-to union theorem.

Progress set union theorem. Let C be a progress set for (F,t,q).

$$[q \Rightarrow q'] \wedge q' \in C \tag{42}$$

$$(\forall c : c \in C : [\neg q \wedge c \Rightarrow awp.G.c]) \tag{43}$$

$\Rightarrow$

$$(wlt.F.q') \wedge t \rightsquigarrow_{F\|G} q'. \tag{44}$$

5. Some progress sets

In this section, we give several examples of how progress sets can be defined.

The set of all predicates. The set of all predicates on the state space of F is a progress set for $(F, true, q)$. Then (43) is equivalent to $\neg q$ being a fixed point of G, so that we get another proof of Misra's fixed point union theorem.

The set of all predicates is primarily of interest because it demonstrates the existence of a progress set for every program and predicate.

The set of all stable predicates. The set of all stable predicates of *any* program on the appropriate state space is closed under arbitrary conjunction (33) and disjunction (34) and is therefore a candidate for progress sets.

Rao [19] gave two union theorems for leads-to based on the notion of decoupling and weak decoupling in terms of stability. His results are

Rao's Decoupling Theorem

$$
\begin{array}{l}
\quad p \rightsquigarrow_F q \\
\quad stable.G.q \\
\quad F\ dec_{safe} G \\
\Rightarrow \\
\quad p \rightsquigarrow_{F\|G} q,
\end{array}
$$

where

$$F\ dec_{safe}\ G \equiv (\forall s, r : s \in S_F \wedge stable.G.r : stable.G.s.r)$$

and

Rao's Weak Decoupling Theorem

$$
\begin{array}{l}
\quad p \rightsquigarrow_F q \\
\quad stable.F\|G.q \\
\quad F\ wdec_{safe}\ G \\
\Rightarrow \\
\quad p \rightsquigarrow_{F\|G} q,
\end{array}
$$

where

$$F\ wdec_{safe}\ G \equiv F\ dec_{safe}\ F\|G.$$

Both of these theorems are simple corollaries of the progress set union theorem. For the decoupling theorem, let $C.F.q = \{r | stable.G.r\}$. From $F\ dec_{safe}\ G$ and $stable.G.q$, $C.F.q$ is a progress set for $(F, true, q)$. Since $stable.G.r \Rightarrow r \wedge \neg q\ co_G\ r$, the theorem follows. For the weak decoupling theorem, let $C.F.q = \{r | stable.F\|Gr\}$. From $F\ wdec_{safe}\ G$, and $stable.F\|Gq$, $C.F.q$ is a progress set for $(F, true, q)$. Since $stable.F\|Gr \Rightarrow r \wedge \neg q co_G\ r$, the theorem follows.

Rao used these results to explore notions of commutativity that allow compositional progress results rather than advocating their direct use in programming. Indeed, direct use would seem to be counterproductive since the theorems themselves are rather noncompositional, requiring detailed knowledge of both F and G in order to determine whether they are decoupled or weakly decoupled.

On the other hand, our more general theorem can be used in a similar, but more "compositional" way where F and G can be decoupled via a third program G'. Instead of checking whether two programs F and G are decoupled or weakly decoupled, given F, we *choose* a program G' so that the set of stable predicates of G' is a progress set for F. Ideally, G' is chosen so that its set of stable predicates has a simple structure and is easily described. Then, to compose a program G with F, it is only necessary to check that G satisfy (43) for the stable predicates of G'. It suffices that G is refinement of G',

thus the stable predicates of G' are stable in G, and we have the following corollary of the progress set union theorem.

Decoupling via G' union theorem. Let the set of all stable predicates of G' be a progress set for $(F, true, q)$.

$$G' \leqslant G$$
$$\Rightarrow$$
$$wlt.F.q \rightsquigarrow_{F\|G} q. \tag{45}$$

It is worth noting that when the set of stable predicates of G is taken as a progress set, C, then $cl.C.r$ is the strongest stable predicate weaker than r in G, or $sst.G.r$ [20]. The set of states of G that are reachable from r is given by $sst.G.r$.

Additional program properties that generate potential progress sets. In the previous section, we discussed using the set of all stable properties of some program G as a potential progress set for (F, t, q). In this section, we give three more program properties that are slightly weaker than stable such that all predicates satisfying the property for some program G are closed under conjunction and disjunction and are therefore potential progress sets for (F, t, q). (We still need to check the remaining conditions on progress sets.) Like the set of all stable predicates, all of the predicates so obtained satisfy (43) for G.

Stable.G-not-leaving-q

$$\{r | r \wedge \neg q \ co_G \ r \wedge [r \wedge q \ co_G \ r \vee \neg q]\}. \tag{46}$$

In this case, $cl.C.r$ is the set of states that are reachable from r along a sequence of states satisfying the requirement that once q holds for some state in the sequence, it holds for all later states in the sequence.

For two states σ and γ connected by such a sequence we write σ Reach.G.nl.q γ.

Stable.G-not-leaving-q-all-directions

$$\{r | r \wedge (\neg q \vee awp.q) \ co_G \ r\}. \tag{47}$$

Here, $cl.C.r$ is the set of states that are reachable from r via a sequence of states where once q holds, the sequence cannot be extended without maintaining q.

Stable.G-outside-q

$$\{r | r \wedge \neg q \ co_G \ r\}. \tag{48}$$

In this latter case, $cl.C.r$ is the set of states that are reachable from r via a sequence of states where q holds for at most the final state in the sequence.

Also, note that for each of these choices for C, $q \in C$ trivially, thus providing alternatives to the set of all stable predicates, which requires that q be stable in G. For later use we also note that a predicate p with $[q \Rightarrow p]$ satisfies (46) and (47) if it satisfies (48).

Progress sets from a relation. Let R be a reflexive, transitive relation on the state space and σ and γ be representative states. Then the set of predicates r such that

$$(\forall \sigma, \gamma : r.\sigma \wedge \sigma R \gamma \Rightarrow r.\gamma) \tag{49}$$

is closed under arbitrary conjunction and disjunction.[2]

5.1. Composition theorems based on monotonicity and commutativity

The previous section listed several ways that sets of predicates closed under arbitrary conjunction and disjunction can be generated. In this section, we give two theorems that are helpful in showing (40), repeated here for convenience, under the assumption of (33, 34, 38, and 39).

$$(\forall m, s : [q \Rightarrow m], s \in F : (t \wedge m) \in C \Rightarrow (t \wedge (q \vee s.m)) \in C) \tag{50}$$

Commutativity of $cl.C$ and command. Assuming (33, 34, 38, and 39), the following implies (40):

$$(\forall s : s \in S_F : (\forall m : [q \Rightarrow m] : [cl.C.(t \wedge s.m) \Rightarrow t \wedge (q \vee s.(cl.C.(t \wedge m)))])) \tag{51}$$

Proof.

$$\begin{array}{ll} & [q \Rightarrow m] \wedge t \wedge m \in C \\ \Rightarrow & \quad \{(51) \text{ and } cl.C.t \wedge m = t \wedge m\} \\ & [cl.C.t \wedge s.m \Rightarrow t \wedge (q \vee s.(t \wedge m))] \\ \Rightarrow & \quad \{\text{monotonicity of } s\} \\ & [cl.C.t \wedge s.m \Rightarrow t \wedge (q \vee s.m)] \\ \Rightarrow & \quad \{\text{disjunction of left and right side with } t \wedge q \\ & \quad \text{which is equal } cl.C.t \wedge q\} \\ & [cl.C.t \wedge q \vee cl.C.t \wedge s.m \Rightarrow t \wedge (q \vee s.m)] \\ \Rightarrow & \quad \{\text{disjunctivity of } cl.C\} \\ & [cl.C.t \wedge (q \vee s.m) \Rightarrow t \wedge (q \vee s.m)] \\ \Rightarrow & \quad \{\text{def. of closed}\} \\ & t \wedge (q \vee s.m) \in C. \end{array}$$

In the next theorem, we require that the command corresponding to each s is deterministic and given by a functional state transformer f_s.

[2] If we let R be defined by $\sigma R \gamma = (cl.C.\langle \sigma \rangle).\gamma$, where $\langle \sigma \rangle$ is the point predicate that holds at σ then the set of predicates generated by R is just C.

Commutativity of functions and relation. Let R be defined by $\sigma R\gamma = (cl.C.\langle\sigma\rangle).\gamma$. Then the condition below implies (51).

$$(\forall s : s \in S_F : (\forall \sigma, \gamma \in \text{state space of } F\|G : \sigma\ R\gamma \wedge t.\sigma \Rightarrow q.\sigma \vee q.\gamma \vee (f_s.\sigma\ R\ f_s.\gamma))). \tag{52}$$

Proof.

$$\begin{array}{ll} & [q \Rightarrow m] \wedge (cl.C.t \wedge s.m).\gamma \\ \Rightarrow & \quad \{\text{definition of } R\} \\ & \exists\sigma : \sigma R\gamma : (t \wedge s.m).\sigma \\ \Rightarrow & \quad \{\text{rewritten with } f_s \text{ using } \textit{stable } F.t\} \\ & (t \wedge m).f_s.\sigma \\ \Rightarrow & \quad \{(52) \Rightarrow (q.\sigma \vee q.\gamma \vee f_s.\sigma\ R\ f_s.\gamma)\} \\ & q.\sigma \vee q.\gamma \vee (cl.Ct \wedge m).f_s.\gamma \\ \Rightarrow & \quad \{q \in C \text{ and } \sigma R\gamma\} \\ & q.\gamma \vee (cl.C.t \wedge m).f_s.\gamma \\ \Rightarrow & \quad \{\text{rewritten with } s \text{ instead of } f_s\} \\ & q.\gamma \vee (s.cl.C.t \wedge m).\gamma \\ \Rightarrow & \quad \{t \in C \text{ implies } t.\sigma \Rightarrow t.\gamma\} \\ & (t \wedge (q \vee s.cl.C.t \wedge m)).\gamma. \end{array}$$

5.2. Monotonicity

This section gives a composition theorem based on monotonicity with respect to a partial order $\leqslant$.

Again, we require that the command corresponding to each $s \in S_F$ and $t \in S_G$ is deterministic and given by a functional state transformer f_s and g_t, respectively.

We use the following definitions.

A predicate q is called monotonic with respect to $\leqslant$ if

$$(\forall \sigma, \gamma : q.\sigma \wedge (\sigma \leqslant \gamma) \Rightarrow q.\gamma). \tag{53}$$

A function f is called monotonic with respect to $\leqslant$ if

$$(\forall \sigma, \gamma : (\sigma \leqslant \gamma) \Rightarrow f.\sigma \leqslant f.\gamma). \tag{54}$$

A function f is called nondecreasing with respect to $\leqslant$ if

$$(\forall \sigma : \sigma \leqslant f.\sigma). \tag{55}$$

If we take the set of all monotonic predicates as C, then the condition $(\forall s : s \in S_F :$ f_s is monotonic with respect to $\leqslant)$ implies (52) and $(\forall s : s \in S_G : g_t$ is non-decreasing with respect to $\leqslant)$ implies that all predicates in C are stable and satisfy therefore (43), so that we have the following corollary:

For a partial order $\leqslant$

$$\begin{aligned} & q \text{ is monotonic with respect to } \leqslant \\ & \quad (\forall s : s \in S_F : f_s \text{ is monotonic with respect to } \leqslant) \\ & (\forall s : s \in S_G : g_s \text{ is non-decreasing with respect to } \leqslant) \\ \Rightarrow & \\ & wlt.F.q \rightsquigarrow_{F\|G} q. \end{aligned}$$

This result can be applied in many programming situations. One example is PCN, see for example [8], processes communicate via so-called definitional variables. A definitional variable is initially undefined and may have assigned a value at most once. We can express this as monotonicity with respect to the partial order given by $(\sigma \leqslant \gamma) \equiv (\sigma \text{ undefined } \vee \sigma = \gamma)$. Another example are processes that communicate by message passing where the partial order is given by the length of the messages that have been sent along a communication channel.

6. Generalized commutativity conditions

The importance of commutativity in program composition has been known for some time. Both Lipton [14] and Misra [16] have proposed relevant conditions and their relationship has been explored by Rao [19]. Here, we give generalized definitions of both Lipton and Misra commutativity and prove a composition theorem using these results. The advantage of our results is that they apply when q in $p \rightsquigarrow q$ is not stable.

The commutativity conditions are conditions on functions, we therefore assume that all commands are deterministic and are expressed by functional state transformers f_s for program F and g_s for program G. In addition, we assume that for each function, there is a guard predicate written b_s or $[\,]f_s$, and if the guard predicate is false, then the state is unchanged, i.e.

$$[\neg b_s \Rightarrow (f_s = id)],$$

where id is the identity function.

If $b_s.\sigma$, then we say that f_s is enabled at σ and write this as $[\sigma]f_s$.

Now we assume that we have a second function g_t with guard c_t. The composition of functions $f_s g_t$ is evaluated from the right, i.e. $f_s g_t.\sigma = f_s.(g_t.\sigma)$ and we define enabled for the composition as

$$[\sigma]f_s g_t = c_t.\sigma \wedge b_s.g_t.\sigma. \tag{56}$$

Left Lipton commutativity outside q. Lipton proposed a commutativity condition where left commutativity can be briefly stated as: If for all states, $[\sigma]fg$, then $[\sigma]gf$ and $f.g.\sigma = g.f.\sigma$, i.e. if fg is enabled, then so is gf and both give the same result.

Here, we give a definition that applies to states where some predicate q does not hold. For two guarded functions f and g left Lipton commutativity outside q is given by:

$$(f \,\mathrm{lco}_{\ell}.\mathrm{q}\, g) \equiv$$
$$(\forall \sigma : \neg q.\sigma \wedge [\sigma] fg \wedge \neg q.g.\sigma : [\sigma] gf \wedge f.g.\sigma = gf.\sigma \wedge (q.f.\sigma \Rightarrow q.fg.\sigma)). \quad (57)$$

Operationally left Lipton commutativity outside q can be described as follows: If σ is outside q, i.e. if $\neg q.\sigma$ and fg is enabled and $g.\sigma$ is outside q, then gf is enabled and $fg = gf$ and if the result $fg.\sigma$ is outside q then $f.\sigma$ is outside q. Note that if $q = false$ we get the original definition given in [14, 19].

Left Lipton commutativity is extended to programs by requiring all pairs of functions from the programs to commute. For two programs F and G:

$$(F \;\mathrm{lco}_{\ell}.\mathrm{q}\; G) \equiv (\forall f_s, g_t : s \in S \wedge t \in T : f_s \;\mathrm{lco}_{\ell}.\mathrm{q}\; g_t). \quad (58)$$

Misra commutativity outside q. Misra defined a slightly different commutativity condition. Two functions f and g Misra commute if at points when both are enabled, both compositions are enabled and give the same result. As above, we give a modified condition that applies to states where a predicate q does not hold.

For two guarded functions f and g:

$$(f \;\mathrm{co}_{m}.\mathrm{q}\; g) \equiv$$
$$(\forall \sigma : \neg q.\sigma \wedge [\sigma] f \wedge [\sigma] g : [\sigma] fg \wedge [\sigma] gf \wedge fg.\sigma = gf.\sigma \wedge$$
$$(q.f.\sigma \vee q.g.\sigma \Rightarrow q.fg.\sigma \vee q.f.\sigma \wedge q.g.\sigma)). \quad (59)$$

Operationally this can be described as follows: If σ is outside q and f and g are enabled, then fg and gf are enabled and $fg = gf$ and if the result $fg.\sigma$ is outside q and if f or g is outside q at σ, then both are. If $q = false$, then we get the original definition of [16].

Misra commutativity is extended to programs in the obvious way. For two programs F and G:

$$(F \;\mathrm{co}_{m}.\mathrm{q}\; G) \equiv (\forall f_s, g_t : s \in S \wedge t \in T : f_s \mathrm{co}_{m}.\mathrm{q} g_t). \quad (60)$$

Now we show the main result of this section, if we have Left Lipton and Misra commutativity outside q that the set of predicates that are *stable.G-not-leaving-q* (46) are a progress set for (*F.true.q*).

Commutativity outside q and *stable.G-not-leaving-q.*

$$(F \;\mathrm{lco}_{\ell}.\mathrm{q}\; G) \wedge (F \;\mathrm{co}_{m}.\mathrm{q}\; G)$$

$\Rightarrow$

The set of all predicates that are *stable.G-not-leaving-q*
is a progress set for (*F.true.q*).

Proof. Since q is *stable.G-not-leaving-q*, it is sufficient to show that (40) holds, or the stronger (52) with relation *Reach.G.nl.q*, (46), i.e.

$$(\forall s : s \in S : \sigma \text{ Reach.G.nl.q } \gamma \Rightarrow q.\sigma \vee q.\gamma \vee (f_s.\sigma \text{ Reach.G.nl.q } f_s.\gamma))$$

(1) $q.\gamma$

$$\begin{array}{ll} & (\sigma \text{ Reach.G.nl.q } \gamma) \wedge q.\gamma \\ \Rightarrow & \\ & q.\sigma \vee q.\gamma \vee (f_s.\sigma \text{ Reach.G.nl.q } f_s.\gamma), \end{array}$$

(2) $\neg q.\gamma$

(2a)

$$\begin{array}{ll} & (\sigma \text{ Reach.G.nl.q } \gamma) \wedge (\sigma \neq \gamma) \wedge \neg q.\gamma \\ \Rightarrow & \quad \{(46)\} \\ & \exists \sigma = \sigma_0, \sigma_1, \ldots, \sigma_n = \gamma :: (\forall i : 0 \leqslant i < n : \exists t : t \in T : \sigma_{i+1} = g_t.\sigma_i) \\ & \wedge (\forall i : 0 \leqslant i < n : q.\sigma_i \Rightarrow q.\sigma_{i+1}) \\ \Rightarrow & \quad \{\text{omitting all } \sigma_i \text{ with } \sigma_{i+1} = \sigma_i\} \\ & \exists \sigma = \sigma_0, \sigma_1, \ldots, \sigma_n = \gamma :: (\forall i : 0 \leqslant i < n : \exists t : t \in T : [\sigma_i] g_t \wedge \sigma_{i+1} = g_t.\sigma_i) \\ & \wedge (\forall i : 0 \leqslant i < n : q.\sigma_i \Rightarrow q.\sigma_{i+1}) \\ \Rightarrow & \quad \{\neg q(\gamma)\} \\ & \exists \sigma = \sigma_0, \sigma_1, \ldots, \sigma_n = \gamma :: (\forall i : 0 \leqslant i < n : \exists t : t \in T : [\sigma_i] g_t \wedge \sigma_{i+1} = g_t.\sigma_i) \\ & \wedge (\forall i : 0 \leqslant i \leqslant n : \neg q.\sigma_i). \end{array}$$

(2b)

$$\begin{array}{ll} & s \in S \\ \Rightarrow & \quad \{\text{construction of a set for } f_s\} \\ & (i : 0 \leqslant i \leqslant n : \sigma'_i = f_s.\sigma_i \quad \text{if } f_s \text{ is enabled for } \sigma_i,\ \sigma'_i = \sigma_i \text{ otherwise}). \end{array}$$

(2c) It is now sufficient to prove:

$$\forall i : 0 \leqslant i < n : \exists t : t \in T : \sigma'_{i+1} = g_t.\sigma'_i \wedge (q.\sigma'_i \Rightarrow q.\sigma'_{i+1})$$

(2c1) $\sigma'_{i+1} \neq \sigma_{i+1}$

$$\begin{array}{ll} & \sigma'_{i+1} \neq \sigma_{i+1} \\ \Rightarrow & \quad \{f_s \text{ must be enabled at } \sigma_{i+1}\} \\ & [\sigma_{i+1}] f_s \wedge \sigma'_{i+1} = f_s.\sigma_{i+1} \\ \Rightarrow & \quad \{\text{By (2a) } \neg q(\sigma_i) \wedge [\sigma_i] g_t \wedge \neg q.\sigma_{i+1} \wedge \sigma_{i+1} = g_t.\sigma_i\} \\ & \neg q.\sigma_i \wedge [\sigma_i] f_s g_t \wedge \neg q.g_t.\sigma_i \wedge \sigma'_{i+1} = f_s g_t.\sigma_i \\ \Rightarrow & \quad \{F \text{ lco}_\ell\text{.q } G\} \\ & [\sigma_i] g_t f_s \wedge g_t f_s.\sigma_i = f_s g_t.\sigma_i \wedge (q.f_s.\sigma_i \Rightarrow q.g_t f_s.\sigma_i) \\ \Rightarrow & \quad \{\sigma'_{i+1} = f_s g_t.\sigma_i\} \\ & [\sigma_i] g_t f_s \wedge \sigma'_{i+1} = g_t f_s.\sigma_i \\ \Rightarrow & \quad \{[\sigma_i] g_t f_s \Rightarrow [\sigma_i] f_s, \text{def.} \sigma'_i\} \\ & \sigma'_{i+1} = g_t.\sigma'_i \wedge (q.\sigma'_i \Rightarrow q.\sigma'_{i+1}). \end{array}$$

(2c2) $\sigma'_{i+1} = \sigma_{i+1} \wedge \sigma'_i \not\models \sigma_i$

$$
\begin{array}{ll}
 & \sigma'_i \neq \sigma_i \\
\Rightarrow & \{f_s \text{ must be enabled at } \sigma_i, \text{ also by (2a)}, g_t \text{ is enabled at } \sigma_i\} \\
 & \neg q(\sigma_i) \wedge [\sigma_i] f_s \wedge \sigma'_i = f_s.\sigma_i \wedge [\sigma_i] g_t \wedge \neg q.g_t.\sigma_i \\
\Rightarrow & \{F \text{ co}_m\text{.q } G\} \\
 & [\sigma_i] f_s g_t \wedge [\sigma_i] g_t f_s \wedge f_s g_t.\sigma_i = g_t f_s.\sigma_i \wedge (q.f_s.\sigma_i \Rightarrow q.g_t f_s.\sigma_i) \\
\Rightarrow & \{[\sigma_i] f_s g_t \Rightarrow [g_t.\sigma_i] f_s\} \\
 & [\sigma_{i+1}] f_s \wedge f_s.\sigma_{i+1} = g_t.\sigma'_i \\
\Rightarrow & \{\text{def. of } \sigma'_{i+1}\} \\
 & \neg q.\sigma'_i \wedge \sigma'_{i+1} = g_t.\sigma'_i \wedge (q.\sigma'_i \Rightarrow q.\sigma'_{i+1}).
\end{array}
$$

(2c3) $\sigma'_{i+1} = \sigma_{i+1} \wedge \sigma'_i = \sigma_i$

By (2a) $\sigma'_{i+1} = g_t.\sigma'_i \wedge (q.\sigma'_i \Rightarrow q.\sigma'_{i+1})$.

Now we can state the theorem on commutativity, which follows now from the progress set union theorem.

Commutativity outside q theorem.

$$
\begin{array}{l}
p \leadsto_F r \\
\neg q \wedge r \text{ co.G } r \vee q \\
(F \text{ lco}_\ell\text{.q } G) \wedge (F \text{ co}_m.q\ G) \\
\Rightarrow \\
p \leadsto_{F \| G} r \vee q.
\end{array}
\tag{61}
$$

The next example applies the theorem to a simple handshaking protocol. The results of Rao are not applicable here since the target predicate is not stable.

Example. *Consumer and producer.* Variable x is a local variable of F, y of G. Both programs share a one element buffer b.

F: Consumer

consume : $x, b := b, \perp$ if $b \neq \perp$
additional assignments that do not modify b
or any variable of G.

G: Producer

produce : $b := y$ if $b = \perp$
additional assignments that do not modify b
or any variable of F.

Now $b = k \leadsto_F b = \perp$. We want to apply the theorem to show $b = k \leadsto_{F \| G} b = \perp$.

With $q = (b = \bot)$, the conditions of the theorem (61) hold:

(1) $\neg q \wedge r$ co.G $q \vee r$ holds because in our case $r = q$.

(2) F lco$_\ell$.q G and F co$_m$.q G:

Since there is no interaction between between F and G except via b and the value of q is changed only by *consume* and *produce*, we need only look at pairs of functions involving *consume* and *produce*.

The conditions $\neg q.\sigma \wedge [\sigma] f g \wedge \neg q.g.\sigma$ or $\neg q.\sigma \wedge [\sigma] f \wedge [\sigma] g$ never hold if $g =$ *produce*, so that the case with $f =$ *consume* and $g \neq$ *produce* remains.

The commutativity in this case follows because there is no interaction between *consume* and g and $q.consume.g$ holds.

7. Comparison with other work

Another common approach to compositionality (for example, see any of [1, 2, 4–6, 9, 13, 14, 18, 23]) is to specify programs with 2-part properties variously called rely/guarantee, hypothesis/conclusion, assumption/commitment, offers/using, assumption/guarantee, or *guarantees*. The common idea is that assumptions about the environment form part of the specification of a component. The focus of most of these works is to provide proof rules allowing a rely/guarantee property to be proved for a component, and for the rely/guarantee specification of, say $F \| G$, to be obtained from the (rely/guarantee) specifications of F and G. Care must be taken to deal with the circularity introduced when the "environment" is some component, say G, also specified with a rely guarantee property and the behavior of G in $F \| G$ also depends on G's environment, which includes F.

In [12, 13], for example, this problem is dealt with by considering layered systems, and only allowing the rely part of a specification to depend on modules at lower layers. Rules for rely/guarantee properties for UNITY with local variables have been given in [5, 6]. They define properties of the form

$$F \textit{ sat } P \textit{ w.r.t. } R,$$

where F is a program, P is a UNITY property such as leads-to, and R is an *interference predicate* constraining the next state relation of the environment. Two components *cooperate* with respect to their interference predicates if neither violates the interference predicate of the other. Proof rules allow rely/guarantee properties of compositions that cooperate with respect to the interference predicates to be derived from the rely/guarantee properties of the components. In other words, the "rely" part of a specification is always a constraint on the next-state relation of the environment.

Treatments of rely/guarantee specifications in temporal logic frameworks are given, for example, in [2, 18, 23]. The interpretation of a rely/guarantee specification of F is "For all computations:: if a computation satisfies the rely property up to some point, and the next step is a step taken by F, then the guarantee property will be satisfied at

least one more step". In [2], the notation

$$E \xrightarrow{+} M$$

is introduced for this temporal property. Note that this property only makes sense when E is a safety property. A proof rule allows the determination of rely/guarantee properties of systems solely from the rely/guarantee properties of components.

In [4], the interpretation of P *guarantees* Q for temporal properties P and Q is that in any system containing F, if all computations satisfy P then all computations will satisfy Q.

In all of these approaches, the "guaranteed" property can typically be an arbitrary property expressible in some programming logic, and for any component, one is only interested in dealing with a small set of properties.

Finally, we mention that our theorem on decoupling via G' (45) is similar to the approach taken in [22] in that one verifies a property of a program composed with an abstract environment, then later proves that the real environment is a refinement of the abstract one.

In this paper, we are concerned with understanding how the environment can be constrained specifically in order for the leads-to properties of a program to continue to hold in a system containing that program. We give several kinds of constraints and different ways of describing them with the goal of finding constraints that are weak, and easy to check. We unified and generalized previous results along these lines.

8. Conclusions

Starting from "first principles", we gave a general composition theorem for leads-to, then generalized it to a theorem based on the notion of a progress set. Progress sets proved to be an extremely useful device – by choosing different definitions of progress sets, we were able to obtain several different theorems for composing programs without invalidating leads-to properties.

References

[1] M. Abadi, L. Lamport, Composing specifications, ACM Trans. Program. Languages Systems 15 (1993) 73–132.
[2] M. Abadi, L. Lamport, Conjoining specifications, ACM Trans. Program. Languages Systems 17 (1995) 507–534.
[3] K.M. Chandy, J. Misra, Parallel Program Design, a Foundation, Addison-Wesley, Reading, MA, 1988.
[4] K.M.Chandy, B.A. Sanders, Predicate transformers for reasoning about concurrent computation, Sci. Computer Program. 24 (1995) 129–147.
[5] P. Colette, Composition of assumption-commitment specifications in a UNITY style, Sci. Comput. Program. 23 (1994) 107–125.
[6] P. Colette, E. Knapp, Logical foundations for compositional verification and development of concurrent programs in UNITY, 4th Int. Conf. on Algebraic Methodology and Software Technology, 1995, Lecture Notes in Computer Science, vol. 936, Springer, Berlin, 1995.

[7] E.W. Dijkstra, C.S. Scholten, Predicate Calculus and Program Semantics, Springer, Berlin, 1990.
[8] Foster, Olson, Tuecke, Productive parallel programming: the PCN approach, to appear in Scientific Programming.
[9] C.B. Jones, Tentative steps toward a development Method for interfering programs, ACM Trans. Program. Languages Systems 5 (1983) 596–619.
[10] C.S. Jutla, E. Knapp, J.R. Rao, A predicate transformer approach to semantics of parallel programs, Proc. 8th ACM symp. on Principles of Distributed Computing, 1989, pp. 249–263.
[11] E. Knapp, A predicate transformer for progress, Inform. Process. Lett. 33 (1989/90) 323–330.
[12] S.S. Lam, A.U. Shankar, Specifying modules to satisfy interfaces – a state transition approach, Distributed Comput. 6 (1992) 39–63.
[13] S.S. Lam, A.U. Shankar, A theory of interfaces and modules 1: composition theorem, IEEE Trans. Software Eng. 20 (1994) 55–71.
[14] R.J. Lipton, Reduction: a method of proving properties of parallel programs, Commun. ACM 18 (1975) 717–721.
[15] J. Misra, A program-composition theorem involving a fixed-point, Notes on UNITY, pp. 28–91.
[16] J. Misra, Loosely coupled processes, Proc. PARLE'91, Parallel Architectures and Languages Europe, Eindhoven, The Netherlands. Lecture Notes in Computer Science, vol. 506, Springer, Berlin, 1991, pp. 1–26.
[17] J. Misra, A logic for concurrent programming: safety, J. Computer and Software Eng. 3(2) (1995) 239–272.
[18] A. Pnueli, In transition from global to modular temporal reasoning about Programs, Logics and Models of Concurrent Systems, Springer, Berlin, 1985, pp. 123–144.
[19] J.R. Rao, Extensions of the unity methodology: compositionality, fairness and probability in parallelism, Lecture Notes in Computer Science, vol. 908, Springer, Berlin, 1995.
[20] B.A. Sanders, Eliminating the substitution axiom from UNITY logic, Formal Aspects Comput. 3 (1991) 189–205.
[21] B.A. Sanders, Refinement of mixed specifications: a Generalization of UNITY, Acta Informatica 35 (1998) 91–129.
[22] N. Shankar, Lazy Compositional Verification, COMPOS'97, 1997.
[23] E.W. Stark, A proof technique for rely/guarantees properties, Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, vol. 206, Springer, Berlin, 1985, pp. 369–391.