Laws of Parallel Synchronised Termination*

David Sands
dave@diku.dk
DIKU, University of Copenhagen
Universitetsparken 1,
DK-2100 København Ø, Denmark.

Abstract

The salient feature of the composition operators for Gamma programs is that for termination, the parallel composition operator demands that its operands must terminate synchronously. This paper studies the inequational partial correctness properties of the combination of sequential and parallel composition operators for Gamma programs, provable from a particular compositional semantics (Brookes-style transition traces) and shows that the "residual program" input-output laws originally described by Hankin *et al.* are also verified by the model.

1 Introduction

The Gamma Model The Gamma formalism was proposed by Banâtre and Le Métayer [3] as a means for the high level description of parallel programs with a minimum of explicit control. Gamma is a minimal language based on local rewriting of a finite multiset (or *bag*), with an appealing analogy with the chemical reaction process. As an example, a program that sorts an array $a_0, \ldots, a_n$ (of integers, say) could be defined as follows. Represent the array as a multiset of pairs $\{\!|(0, a_0), \ldots, (n, a_n)|\!\}$, then just specify a single rule: "exchange ill-ordered values"

$$sort_A : ((i, x), (j, y) \rightarrow (i, y), (j, x) \Leftarrow i < j \ \& \ x > y).$$

$i < j \ \& \ x > y$ specifies a property (a *reaction condition*) to be satisfied by the selected elements (i, x) and (j, y); these elements are replaced in the multiset (the *chemical solution*) by the elements $(i, y), (j, x)$ (the product of the reaction). Nothing is said in this definition about the order of evaluation of the comparisons; if several disjoint pairs of elements satisfy the reaction condition the comparisons and replacements can even be performed in parallel.

The computation terminates when a stable state is reached, that is to say when no elements of the multiset satisfy the reaction condition (or in general, any of a number of reaction conditions). The interested reader may find a long

*To Appear: Proceedings of the First Imperial College, Department of Computing, Workshop on Theory and Formal Methods, Isle of Thorns, March 1993, Springer-Verlag Workshops in Computer Science

series of examples illustrating the Gamma style of programming in [3]. The benefit of using Gamma in systematic program construction in the Dijkstra-Gries style are illustrated in [2].

A Calculus of Gamma Programs For the sake of modularity it is desirable that a language offers a rich set of operators for combining programs. It is also fundamental that these operators enjoy a useful collection of algebraic laws in order to make it possible to reason about composed programs. Hankin, Le Métayer and Sands [6] defined two composition operators for the construction of Gamma programs from basic reactions, namely sequential composition $P_1 \circ P_2$ and parallel composition $P_1 + P_2$. The intuition behind $P_1 \circ P_2$ is that the stable multiset reached after the execution of P_2 is given as argument to P_1. On the other hand, the result of $P_1 + P_2$ is obtained (roughly speaking) by executing the reactions of P_1 and P_2 (in any order, possibly in parallel), terminating only when neither can proceed further—in the spirit of the original Gamma programs.

As a further simple example, consider the following program which sorts a multiset of integers:

$$\begin{array}{ll} sort_B : match \circ init & \\ \mathsf{where} \quad init : & (x \rightarrow (0, x) \Leftarrow integer(x)) \\ \quad match : & ((i, x), (i, y) \rightarrow (i, x), (i + 1, y) \Leftarrow x \leq y) \end{array}$$

The reaction *init* gives each integer an initial rank of zero. When this has been completed, *match* takes any two elements of the same rank and increases the rank of the larger.

In fact we get a sort program for a multiset of integers by placing $sort_A$ in parallel with $sort_B$ as $sort_A + sort_B$.

Hankin *et al.* [6] derived a number of program refinement and equivalence laws for parallel and sequential composition, by considering the input-output behaviour induced by an operational semantics. So, for example, the program $sort_A + sort_B$ which is by definition $sort_A + (match \circ init)$ is refined by the program

$$(sort_A + match) \circ init.$$

This refinement is an instance of a general refinement law:

$$P + (Q \circ R) \geq (P + Q) \circ R,$$

and is one of a number of laws relating parallel and sequential composition. The main shortcoming of such refinement laws from [6] is that there is no guarantee that the refinement and equivalences described are compositional, so that a refinement of a sub-program does not necessarily imply a refinement of the program.

In [10] the author directly addresses this problem by defining a compositional (denotational) semantics for Gamma. A few of the laws from [6] are shown to hold also in this model, and questions of abstractness, and alternative combining forms are studied. In this paper we continue this study of the properties of this language in particular addressing the laws concerning the so-called *residual program*— a synactic characterisation of the program part of the state just before its termination.

In **Section 2** we summarise the operational semantics of Gamma programs, and in **Section 3** we sketch the denotational semantics given in [10] which adapts a method of Brookes [4] for giving a (fully abstract) denotational semantics for a parallel shared variable *while* language. The key technique is to give the meaning of a program as a set of *transition traces* which represent both the computation steps that the program can perform, and the ways that the program can interact with its environment. In **Section 4** we show how the model can verify a number of typical and some not so typical laws of parallel and sequential compositions. In **Section 5** we consider the residual program properties from [6], and show that the residual program notion arises naturally from the laws.

2 Operational Semantics

In this section we consider the operational semantics of programs consisting of basic reactions (written $(A \Leftarrow R)$, where R is the reaction condition, and A is the associated action, both assumed to have the same arity), together with two *combining forms*: sequential composition, $P_1 \circ P_2$, and parallel combination, $P_1 + P_2$ as introduced in [6].

$$P \in \mathbf{P} ::= (A \Leftarrow R) \mid P \circ P \mid P + P$$

To define the semantics for these programs we define a single step transition relation between *configurations*. The *terminal* configurations are just multisets, and the *intermediate* configurations are program, multiset pairs written $\langle P, M\rangle$, where $M \in \mathbf{M}$ is the set of finite multisets of elements. The domain of the elements is left unspecified, but is expected to include integers, booleans and tuples.

We define the single step transitions first for the individual reactions $(A \Leftarrow R)$. A reaction terminates on some multiset exactly when there is no submultiset of elements which (when viewed as a tuple) satisfy the reaction condition. We will not specify the details of the reaction conditions and actions, but assume, as previously, that they are total functions over tuples of integers, truth values, tuples etc.

$$\langle (A \Leftarrow R), M\rangle \to M \quad \text{if } \neg\exists \vec{a} \subseteq M.\, R\vec{a}$$

Otherwise, if we can form a tuple from elements of the multiset, and this tuple satisfies the reaction condition, the selected elements can be replaced by the elements produced by the associated action function:

$$\langle (A \Leftarrow R), M \uplus \vec{a}\rangle \to \langle (A \Leftarrow R), M \uplus A\vec{a}\rangle \quad \text{if } R\vec{a}$$

The *terminal transitions* have the form $\langle P, M\rangle \to M$, and are only defined for programs not containing sequential composition, so the remaining *terminal* transitions are defined by the following synchronised-termination rule:

$$\frac{\langle P, M\rangle \to M \quad \langle Q, M\rangle \to M}{\langle P + Q, M\rangle \to M}$$

The remaining intermediate transitions are given by first defining what we will call *active contexts*.

Definition 1 An *active context*, $\mathbf{A}$ is a term containing a single hole $[\,]$:

$$\mathbf{A} ::= [\,] \mid P + \mathbf{A} \mid \mathbf{A} + P \mid P \circ \mathbf{A}$$

Let $\mathbf{A}[P]$ denote active context $\mathbf{A}$ with program P in place of the hole.

The idea of active contexts is that they isolate parts of a program that can affect the next transition, so that for example, the left-hand side of a sequential composition is not active (but it can become active once the right-hand side has terminated). The remaining transitions are defined by the following two rules:

$$\frac{\langle P, M\rangle \rightarrow \langle P', M'\rangle}{\langle \mathbf{A}[P], M\rangle \rightarrow \langle \mathbf{A}[P'], M'\rangle} \qquad \frac{\langle Q, M\rangle \rightarrow M}{\langle \mathbf{A}[P \circ Q], M\rangle \rightarrow \langle \mathbf{A}[P], M\rangle}$$

The first says that if there is a possible reaction in an active context then it can proceed, while the second says that if the right hand side of a sequential composition can terminate, then it can be erased.

In [6] an equivalent definition is given in terms of the more traditional SOS style, and active contexts are given as a derived construction to facilitate proofs of certain properties.

3 Transition Trace Model

In this study we only consider the simplest of the input-output orderings, that of *partial correctness* (the $\leq_L$ order of [6]).

> We define $P_1 \leq P_2$ whenever, for each possible input M, if P_1 can terminate producing some multiset N then so can P_2.

This partial correctness preorder, and associated equivalence $\sim$ was was shown to satisfy a number of properties in [6] (in fact they are stated for a stronger ordering which also treats nontermination as a possible "result").

Because of the lack of a general substitutivity property, the use of the partial correctness laws in reasoning about programs is limited. As is standard, we are therefore interested in the largest contextual (pre)congruence relation contained in $\leq$. Let $\mathbf{C}$ range over general program contexts, then we define *operational approximation* and *operational congruence* respectively by:

$$\begin{array}{lcl} P_1 \sqsubseteq_o P_2 & \iff & \forall \mathbf{C}.\, \mathbf{C}[P_1] \leq \mathbf{C}[P_2] \\ P_1 \equiv_o P_2 & \iff & P_1 \sqsubseteq_o P_2 \;\&\; P_2 \sqsubseteq_o P_1 \end{array}$$

We will now describe a semantics for programs which is suitable for reasoning about operational approximation and equivalence. It is clear that we must consider intermediate behaviour of programs to obtain a substitutive equivalence. However, it is not enough to consider just the sequences of intermediate states in a program's execution, because they do not take into account the possible *interference* (from the program's surrounding context) that can occur during execution. The solution we adopt is an adaptation of a technique for giving denotational semantics to shared-variable *while* languages proposed by Brookes [4]; related tequniques include Hennessy and Plotkin's *resumptions* [7] and Abrahamson's *move* sequences [1].

Transition Traces We adapt Brooke's transition trace model for our language. We define the meaning of a command as a set of transition traces, and show that the definition can be given compositionally.

Definition 2 The transition trace function $\mathsf{T}[\![\_]\!] : \mathbf{P} \to \wp((\mathbf{M} \times \mathbf{M})^+)$ is given by

$$\begin{array}{rl} \mathsf{T}[\![P]\!] = & \{(M_0, N_0)(M_1, N_1) \ldots (M_k, N_k) | \\ & \langle P, M_0 \rangle \to^* \langle P_1, N_0 \rangle \ \& \\ & \langle P_1, M_1 \rangle \to^* \langle P_2, N_1 \rangle \ \& \ \ldots \ \& \ \langle P_k, M_k \rangle \to^* N_k \} \end{array}$$

The intuition behind the use of transition traces to give meaning to a program is that each transition trace

$$(M_0, N_0)(M_1, N_1) \ldots (M_k, N_k) \in \mathsf{T}[\![P]\!]$$

represents a terminating execution of program P in some context starting with multiset M_0, and in which each of the pairs (M_i, N_i) represents computation steps performed by (derivatives of) P and the adjacent multisets N_{i-1}, M_i represent possible interfering computation steps performed by the context. The partial correctness ordering is recoverable from the transition traces by considering the transition traces of length one:

$$P_1 \leq P_2 \iff \{(M, N) \mid (M, N) \in \mathsf{T}[\![P_1]\!]\} \subseteq \{(M, N) \mid (M, N) \in \mathsf{T}[\![P_2]\!]\} .$$

A key feature of the definition of transition traces (in comparison with related approaches) is that they use the reflexive-transitive closure of the one-step evaluation relation, $\to^*$. An important consequence of this reflexivity and transitivity is that they are closed under certain "stuttering" and "absorption" properties described below. In the following, let ϵ denote the empty sequence. Let α and β range over elements of $(\mathbf{M} \times \mathbf{M})^*$.

Definition 3 A set $T \subseteq \wp((\mathbf{M} \times \mathbf{M})^+)$ is closed under left-stuttering and absorption if it satisfies the following two conditions

$$\text{left-stuttering}\frac{\alpha\beta \in T, \beta \neq \epsilon}{\alpha(M, M)\beta \in T} \qquad \text{absorption}\frac{\alpha(M, N)(N, M')\beta \in T}{\alpha(M, M')\beta \in T}$$

Let $\ddagger T$ denote the left-stuttering and absorption closure (henceforth just closure) of a set T. "Stuttering" represents the fact that we can have an arbitrary interference by the context without any visible steps performed by the program. The concept is well known from Lamport's work on temporal logics for concurrent systems [9]. Notice that we say *left*-stuttering to reflect that the context is not permitted to change the state after the termination of the program. In this way each transition trace of a program charts an interaction with its context, until the point of its termination. The *absorption* property is important because prevents "idle" computation steps from becoming semantically significant, which is a problem with the resumption approach [7].

A Compositional Definition of Transition Traces

We now show that the transition traces of a command can be given a denotational definition, which shows that transition trace semantics can be used to prove operational equivalence.

For the basic reaction-action pairs $(A \Leftarrow R)$, we build transition traces by simply considering all sequences of mediating transitions, followed by a terminal transition. Define the following sets:

$$\begin{array}{rcl} \mathsf{mediators}_{(A \Leftarrow R)} & = & \{(M,N) \mid \langle (A \Leftarrow R), M\rangle \rightarrow^* \langle (A \Leftarrow R), N\rangle\} \\ \mathsf{terminals}_{(A \Leftarrow R)} & = & \{(M,N) \mid \langle (A \Leftarrow R), M\rangle \rightarrow^* N\}\,. \end{array}$$

Sequential composition has an easy definition. We just concatenate the transition traces from the transition traces of the components, and take their closure. Define the following sequencing operation for transition trace sets:

$$T_1 \mathbin{;} T_2 = \{\alpha\beta \mid \alpha \in T_1, \beta \in T_2\}\,.$$

Not surprisingly parallel composition is built with the use of an interleaving combinator. First we define interleaving on single traces: for α and β in $(\mathbf{M},\mathbf{M})^*$ let $\alpha \sharp \beta$ be the set of all their interleavings, given inductively by

$$\begin{array}{rcl} \epsilon \sharp \beta & = & \beta \sharp \epsilon \;=\; \{\beta\} \\ (M,M')\alpha \sharp (N,N')\beta & = & \{(M,M')\gamma \mid \gamma \in \alpha \sharp (N,N')\beta\} \\ & & \cup\, \{(N,N')\gamma \mid \gamma \in (M,M')\alpha \sharp \beta\}\,. \end{array}$$

Now to define the transition traces of $P_1 + P_2$ we must ensure that the traces of P_1 and P_2 are interleaved, but not arbitrarily. The termination step of a parallel composition requires an agreement at the point of their termination. For this purpose, we define the following interleaving operation on transition traces:

$$T_1 \oplus T_2 = \{\alpha(M,M) \mid \alpha_1(M,M) \in T_1, \alpha_2(M,M) \in T_2, \alpha \in \alpha_1 \sharp \alpha_2\}$$

Proposition 4 ([10]) The transition traces of a program are characterised by the following denotational definition:

$$\begin{array}{rcl} \mathsf{T}[\![(A \Leftarrow R)]\!] & = & (\mathsf{mediators}_{(A \Leftarrow R)})^* \mathbin{;} \mathsf{terminals}_{(A \Leftarrow R)} \\ \mathsf{T}[\![P_1 \circ P_2]\!] & = & \ddagger(\mathsf{T}[\![P_2]\!] \mathbin{;} \mathsf{T}[\![P_1]\!]) \\ \mathsf{T}[\![P_1 + P_2]\!] & = & \ddagger(\mathsf{T}[\![P_1]\!] \oplus \mathsf{T}[\![P_2]\!]). \end{array}$$

Define the transition trace ordering on programs $\sqsubseteq_t$ as

$$P_1 \sqsubseteq_t P_2 \iff \mathsf{T}[\![P_1]\!] \subseteq \mathsf{T}[\![P_2]\!].$$

The operations used to build the compositional definition are all monotone with respect to subset inclusion, and so a simple induction on contexts is sufficient to give

$$P_1 \sqsubseteq_t P_2 \Rightarrow \forall \mathbf{C}.\, \mathbf{C}[P_1] \sqsubseteq_t \mathbf{C}[P_2]$$

and since $P \sqsubseteq_t Q$ implies $P \leq Q$, we have the desired characterisation of operational approximation:

$$P_1 \sqsubseteq_t P_2 \Rightarrow P_1 \sqsubseteq_o P_2.$$

However, as we showed in [10] we cannot reverse the implication—the transition trace semantics is not (inequationally) *fully abstract.*

4 Basic Laws

In this section we present a number of the basic laws of synchronised termination.

Transition Trace Properties Certain structural laws follow directly from the properties of the transition trace combinators, which we state without proof.

Proposition 5 Let S_1, S_2, T_1, T_2, etc. range over sets of transition traces not including ϵ (but not necessarily closed). Then we have the following:

$$\begin{array}{rcl} T_1 \oplus T_2 & = & T_2 \oplus T_1 \\ T_1 \oplus (T_2 \oplus T_3) & = & (T_1 \oplus T_2) \oplus T_3 \\ \ddagger(\ddagger T_1 \oplus \ddagger T_2) & = & \ddagger(T_1 \oplus T_2) \end{array}$$

$$\begin{array}{rcl} T_1 \fatsemi (T_2 \fatsemi T_3) & = & (T_1 \fatsemi T_2) \fatsemi T_3 \\ \ddagger(\ddagger T_1 \fatsemi \ddagger T_2) & = & \ddagger(T_1 \fatsemi T_2) \\ \ddagger(T_1 \fatsemi T_2) & = & (\ddagger T_1 \fatsemi \ddagger T_2) \cup (\ddagger T_1 \bowtie \ddagger T_2) \end{array}$$

where $A \bowtie B = \{\alpha(M,N)\beta \mid \alpha(M,N') \in A,\ (N',N)\beta \in B\}$

The "skip" laws The program which "does nothing"—one which can never perform any reactions and therefore can only terminate—will be represented by a single reaction-action pair $(A \Leftarrow False)$. Since the reaction condition is false, the action A, and arity are irrelevant.

In terms of the basic input-ouput partial correctness ordering it is clear that $(A \Leftarrow False)$ obeys the usual "skip" laws of being an identity for sequential and parallel composition. However, in terms of operational approximation one of these identities fails. Namely,

$$(A \Leftarrow False) \circ P \not\equiv_o P$$

The intuition for this is that $(A \Leftarrow False)$ acts as a de-synchroniser for parallel composition: P must synchronise with its context in order to terminate, but with $(A \Leftarrow False) \circ P$, P is allowed to terminate autonomously, leaving $(A \Leftarrow False)$ to synchronise with its context—which it is trivially always able to do. As an example consider the following two atomic programs:

$$P_1 : x \to 0 \Leftarrow x = 1 \qquad P_2 : x \to 1 \Leftarrow x = 0.$$

Notice that, for example, $\langle P_1 + P_2, \{\!|1|\!\}\rangle$ can never terminate, but $\langle((A \Leftarrow False) \circ P_1) + P_2, \{\!|1|\!\}\rangle \to^* \{\!|1|\!\}$, and so $((A \Leftarrow False) \circ P_1) + P_2 \not\sqsupseteq_o P_1 + P_2$.

Note that we cannot prove any inequalities by appealing to the transition traces, because we lack full abstraction. However, the positive statement

$$P_1 + P_2 \sqsubseteq_o ((A \Leftarrow False) \circ P_1) + P_2$$

can be proved via the transition traces. It should be clear that $\mathsf{T}[\![(A \Leftarrow False)]\!] = \ddagger\{(M,M) \mid M \in \mathbf{M}\}$. Now consider a transition trace $\alpha \in \mathsf{T}[\![P]\!]$. Suppose that the last transition in α is (N',N). Since $(N,N) \in \mathsf{T}[\![(A \Leftarrow False)]\!]$ we know that

$$\alpha(N,N) \in (\mathsf{T}[\![P]\!] \fatsemi \mathsf{T}[\![(A \Leftarrow False)]\!]).$$

By absorbtion it follows that

$$\alpha \in \ddagger(\mathsf{T}[\![P]\!] \mathbin{;} \mathsf{T}[\![(A \Leftarrow False)]\!]) = \mathsf{T}[\![(A \Leftarrow False) \circ P]\!].$$

The remaining compositions with the "skip" program are identities, and are provable by simple applications of the closure properties. Many of the following laws involve the "de-synchroniser", $(A \Leftarrow False)$.

The "chaos" laws There are programs that always diverge. Consider any nullary reaction $Chaos = (() \rightarrow A \Leftarrow True)$ which says: if the empty set is a subset of the multiset, replace that subset by A (some arbitrary multiset). Clearly such a reaction is always applicable, and as a consequence for all contexts $\mathbf{C}$ and programs P,

$$Chaos \equiv_t \mathbf{C}[Chaos] \sqsubseteq_t P.$$

More interestingly, the "near-chaos" program consisting of a *unary* reaction $Await_\emptyset = (x \rightarrow x \Leftarrow True)$ terminates only on the empty set, since for any other multiset the unary reaction condition is applicable. This program was used in [10] to show that the transition trace semantics is not fully abstract, by showing that for all programs P, $P \circ Await_\emptyset \sqsubseteq_o Await_\emptyset$, but that $P \circ Await_\emptyset \not\sqsubseteq_t Await_\emptyset$.

Proposition 6 (The sequential laws)

1. $P \circ (Q \circ R) \equiv_t (P \circ Q) \circ R$
2. $P \circ (A \Leftarrow False) \equiv_t P$
3. $P \sqsubseteq_t (A \Leftarrow False) \circ P$
4. $(A \Leftarrow R) \equiv_t (A \Leftarrow R) \circ (A \Leftarrow R)$

The first two laws follow directly from properties of the transition trace combinators. The last of the sequential laws is an instance of a more general law involving residual programs which we consider in the next section.

Proposition 7 (The parallel laws)

1. $P + (Q + R) \equiv_t (P + Q) + R$
2. $P + Q \equiv_t Q + P$
3. $P \equiv_t (A \Leftarrow False) + P$
4. $P \sqsubseteq_t P + P$
5. $(A \Leftarrow R) \equiv_t (A \Leftarrow R) + (A \Leftarrow R)$

PROOF The fourth law has the least straightforward proof, which is given in essentially the same way as the proof for the corresponding input-output property in [6].

It is sufficient to show that the *strict traces*, $\mathcal{ST}[\![P]\!]$ of a program P, defined to be

$$\begin{aligned}\mathcal{ST}[\![P]\!] = \; & \{(M_0, N_0)(M_1, N_1) \ldots (M_k, N_k) | \\ & \langle P, M_0 \rangle \rightarrow \langle P_1, N_0 \rangle \;\&\; \\ & \langle P_1, M_1 \rangle \rightarrow \langle P_2, N_1 \rangle \;\&\; \ldots \;\&\; \langle P_k, M_k \rangle \rightarrow N_k\}\end{aligned}$$

is contained in $\mathsf{T}[\![P + P]\!]$, since the closure of the strict traces yields the transition traces. The proof of this follows from an induction on the length of each

strict transition trace. This uses the obsevation that every non-terminal transition of a program can be expressed by a proof whose last inference has one of two possible forms:

$$\text{active}\frac{\langle (A \Leftarrow R), M\rangle \to \langle (A \Leftarrow R), M'\rangle}{\langle \mathbf{A}[(A \Leftarrow R)], M\rangle \to \langle \mathbf{A}[(A \Leftarrow R)], M'\rangle} \qquad \text{passive}\frac{\langle Q', M\rangle \to M}{\langle \mathbf{A}[Q \circ Q'], M\rangle \to \langle \mathbf{A}[Q], M\rangle}$$

Now it easily follows from the definition of active contexts that every *active* step $\langle R, N\rangle \to \langle R, N'\rangle$ can be matched (in a single step) by $R + R$, and every *passive* step $\langle R, N\rangle \to \langle R', N\rangle$ can be matched by two steps by $R + R$, *viz.*

$$\langle R + R, N\rangle \to \langle R' + R, N\rangle \to \langle R' + R', N\rangle.$$

The details of the induction are then straightforward. (We also have a slightly simpler proof by structural induction, using the second law of the next proposition) □

Proposition 8 (The Parallel-Sequential Laws)

1. $(P + Q) \circ R \sqsubseteq_t P + (Q \circ R)$
2. $(P_1 + P_2) \circ (Q_1 + Q_2) \sqsubseteq_t (P_1 \circ Q_1) + (P_2 \circ Q_2)$
3. $P \circ (Q + R) \sqsubseteq_t (P \circ Q) + (P \circ R)$

PROOF

1. $$\begin{array}{lll} (P + Q) \circ R & \equiv_t & (P + Q) \circ ((A \Leftarrow False) + R) \quad [7(3)] \\ & \sqsubseteq_t & (P \circ (A \Leftarrow False)) + (Q \circ R) \quad [8(2)] \\ & \equiv_t & P + (Q \circ R) \quad [6(2)] \end{array}$$
2. Let $p_1 = \mathsf{T}[\![P_1]\!], q_1 = \mathsf{T}[\![Q_1]\!]$, etc. From the compositional definition, is necessary to show that

   $$\ddagger(\ddagger(q_1 \oplus q_2) \fatsemi \ddagger(p_1 \oplus p_2)) \subseteq \ddagger(\ddagger(q_1 \fatsemi p_1) \oplus \ddagger(q_2 \fatsemi p_2)).$$

   By the properties of the closure operation, it is sufficient to show that

   $$(q_1 \oplus q_2) \fatsemi (p_1 \oplus p_2) \subseteq \ddagger((q_1 \fatsemi p_1) \oplus (q_2 \fatsemi p_2)).$$

   The details are omitted.
3. $$\begin{array}{lll} P \circ (Q + R) & \sqsubseteq_t & (P + P) \circ (Q + R) \quad [7(4)] \\ & \sqsubseteq_t & (P \circ Q) + (P \circ R) \quad [8(2)] \end{array}$$

□

5 Residual Program Laws

In [6] a notion of "residual program" was introduced as a particularly simple syntactic characterisation of the program component of any configuration that is an immediate predecessor of a terminal configuration (multiset).

Definition 9 (Residual Program) The *residual part* of a program P, written $\underline{P}$, is defined by induction on the syntax:

$$\underline{\underline{(A \Leftarrow R)}} = (A \Leftarrow R) \qquad \underline{\underline{P_1 \circ P_2}} = \underline{\underline{P_1}} \qquad \underline{\underline{P_1 + P_2}} = \underline{\underline{P_1}} + \underline{\underline{P_2}}$$

This provides us with a simple (ie. weak) postcondition for programs via the following property:

$$\langle P, M\rangle \rightarrow^* N \iff \langle P, M\rangle \rightarrow^* \langle \underline{\underline{P}}, N\rangle \rightarrow N$$

As a notational convinience, we define the predicate Φ on a program and a multiset to be true if and only if the residual part of the program is terminated with respect to the multiset:

$$\Phi(P, M) \Leftrightarrow \langle \underline{\underline{P}}, M\rangle \rightarrow M.$$

Intuitively, $\Phi(P, M)$ holds if M is a possible result for the program P. The significance of this is that the predicate $\Phi(P, \_)$ can be constructed syntactically by considering (the negations of) the reaction conditions in $\underline{\underline{P}}$.

A number of operational laws involving residual programs where presented in [6]. Here we show that these laws (and some additional ones) can be verified in our compositional model. The intuition behind the fact that laws also hold in the compositional model is that the residual part of a program expresses concisely the termination synchronisation requirement of the program with its context.

The following key result from [6] establishes when sequential composition correctly implements parallel combination:

$$(\forall M.(\Phi(Q, M) \wedge \langle P, M\rangle \rightarrow^* N) \Rightarrow \Phi(Q, N)) \Rightarrow P \circ Q \leq P + Q.$$

This does not hold for operational approximation because in the premise, it is assumed that P takes over from the execution of Q without interruption. An obvious strengthening of this premise gives

Proposition 10

$$(\forall M.\Phi(P, M) \Rightarrow \Phi(Q, M)) \Rightarrow P \circ Q \sqsubseteq_t P + Q$$

PROOF Assuming $(\forall M.\Phi(P, M) \Rightarrow \Phi(Q, M)$, it is sufficient to prove that

$$\mathsf{T}[\![Q]\!] \mathbin{;} \mathsf{T}[\![P]\!] \subseteq \ddagger(\mathsf{T}[\![P]\!] \oplus \mathsf{T}[\![Q]\!])$$

The details are omitted. □

Proposition 11 (Residual-Program Laws)

1. $P \equiv_t \underline{\underline{P}} \circ P$
2. $(P + \underline{\underline{R}}) \circ (Q + R) \sqsubseteq_t (P \circ Q) + R$
3. $(P \equiv_t \underline{\underline{P}}) \Rightarrow (P \equiv_t P + P)$

PROOF The first part is easily proved by noting the following operational property (see [6]):

$$\langle P, M\rangle \rightarrow^* \langle P', M'\rangle \Rightarrow \underline{\underline{P}} = \underline{\underline{P'}}.$$

The details are then straightforward from the operational definition of transition traces.

The second law is a consequence of the first, since

$$\begin{array}{rll} (P + \underline{\underline{R}}) \circ (Q + R) & \sqsubseteq_t & (P \circ Q) + (\underline{\underline{R}} \circ R) \quad [8(2)] \\ & \equiv_t & (P \circ Q) + R \end{array}$$

The last part is a generalisation of 7(5), and hinges on the fact that residual programs cannot contain sequential compositions, so the result follows by a simple induction on the structure of residual programs. □

The following proposition shows that residual programs arise naturally from the laws, by showing that every program is refined by (ie, approximated by) a "product of sums" of basic reactions.

Define $\sum_{i=1}^{n} P_i$ as $P_1 + \cdots + P_n$, and $\prod_{i=1}^{n} P_i$ as $P_1 \circ \cdots \circ P_n$.

Proposition 12 For all programs P, there exists a set of basic reactions

$$\{(A_{ij} \Leftarrow R_{ij}) \mid i \in \{1 \ldots n\}, j \in \{1 \ldots m_i\}\}$$

such that $\prod_i \sum_{j=1}^{m_i} (A_{ij} \Leftarrow R_{ij}) \sqsubseteq_t P$.

PROOF We give a normalising rewrite system which constructs such a product-of-sums.

Consider the law 8(1) $(P+Q) \circ R \sqsubseteq_t P+(Q \circ R)$. Orienting this inequality from right to left gives the rewrite rule

$$P + (Q \circ R) \longrightarrow (P + Q) \circ R$$

from which we form an associative-commutative rewriting system $\longrightarrow_{ac}$ (see eg. [5]), with $+$ as an associative-commutative operator and $\circ$ as a commutative operator. It should be clear that if $Q \longrightarrow_{ac} R$, then

1. $R \sqsubseteq_t Q$, and
2. if $R \not\longrightarrow_{ac}$ then R is a product-of-sums.

For termination we just use a simple polynomial interpretation of terms: interpret parallel composition as multiplication, sequential composition as sum, and the basic reactions as the integer 2. This interpretation respects the *ac*-equivalence, and it is simple to check that the rewrite strictly reduces the size of the corresponding interpretation (which is always positive). □

Now we also have an interesting property of the above rewrite system with respect to residual programs, namely that the leftmost of the products is equal (modulo ac) to the residual of the original program. This can be seen as follows. The rewriting process obtains a program $(\prod_i P_i) \sqsubseteq_t P$ such that each of the P_i is of the form $\sum_j (A_j \Leftarrow R_j)$. Now we make use of the following:

$$\forall C. \underline{\underline{Q_1}} = \underline{\underline{Q_2}} \Rightarrow \underline{\underline{C[Q_1]}} = \underline{\underline{C[Q_2]}}$$

which is proved by a simple induction on contexts.

Now note that the rewrite rule preserves residual programs, and so by the above property, $\underline{\underline{\prod P_i}}$ is equivalent to $\underline{\underline{P}}$ up to associativity and commutativity of $+$. Since each P_i contains no sequential compositions, then $\underline{\underline{P}} = \underline{\underline{\prod P_i}} = P_1$.

More "parallel" sums of products can be produced by using law 8(2) as a rewrite rule. One possible interest of this result is that programs in the form of a product of sums are relatively easy to reason about in the manner of [2], and is likely to simplify the application of the compositional logic of [8], since it localises the parallel compositions.

6 Conclusions

We have presented the (in)equational partial correctness properties of the combination of sequential and parallel composition operators for Gamma programs,

provable from the compositional semantics first presented in [10]. We have shown that almost all of the laws of the (non-monotonic) input-output ordering considered in earlier work [6], including the "residual program" laws, also hold in this model. The equational reasoning provided by the compositional semantics simplifies a number of the earlier proofs, and has revealed an interesting factorisation law which says that every program is refined by a product-of-sums.

Acknowlegements

Thanks to Simon Gay for comments on an earlier draft, and to Fritz Henglein for some useful technical suggestions. This work was supported by the Danish Research Council, DART Project (5.21.08.03).

References

[1] K. Abrahamson. Modal logic of concurrent nondeterministic programs. In *Proceedings of the International Symposium on Semantics of Concurrent Computation*, volume 70, pages 21–33. Springer-Verlag, 1979.

[2] J.-P. Banâtre and D. Le Métayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[3] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *CACM*, January 1993. (INRIA research report 1205, April 1990).

[4] S. Brookes. Full abstraction for a shared variable parallel language. In *Logic In Computer Science*, 1993. (to appear).

[5] N. Dershowitz and J-P. Jouannaud. *Rewrite Systems*, volume B, chapter 15. North-Holland, 1989.

[6] C. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. Research Report DOC 92/22 (28 pages), Department of Computing, Imperial College, 1992. (short version to appear in the Proceedings of the Fifth Annual Workshop on Languages and Compilers for Parallelism, Aug 1992, Springer-Verlag).

[7] M. Hennessy and G. D. Plotkin. Full abstraction for a simple parallel programming lanuage. In *Mathematical Foundations of Computer Science*, volume 74 of *LNCS*, pages 108–120. Springer-Verlag, 1979.

[8] C. Hankin L. Errington and T. Jensen. Reasoning about Gamma programs. In this volume.

[9] L. Lamport. The Temporal Logic of Actions. Technical Report 79, DEC Systems Research Center, Palo Alto, CA, 1991.

[10] D. Sands. A compositional semantics of combining forms for Gamma programs. In *International Conference on Formal Methods in Programming and Their Applications.* Springer-Verlag, 1993.