Pipelining the Molecule Soup: 
A Plumber’s Approach to Gamma

Martin Weichert

Gothenburg University and Chalmers University of Technology, 
Department of Computing Science, S-412 96 Gothenburg, Sweden 
martinw@cs.chalmers.se

Abstract. Gamma is a language based on multiset rewriting aimed at 
separating coordination from computation. The “pipelining” technique 
turns a producer-consumer-type Gamma program with sequential com-
position into a parallel one, preventing the data from erroneously “flow-
ing back” from the consumer to the producer. The resulting parallel 
composition offers more freedom to coordinate the computation and to 
choose an appropriate refinement later on. This paper extends a previ-
ous paper by Hankin, Le Métayer and Sands. It identifies conditions for 
the pipelining transformation of atomic programs and proves it correct 
under these conditions with relational reasoning. It also proves refine-
ment in the other direction correct with respect to a stronger refinement 
relation, statebased simulation, and strengthens it further to a precon-
gruence using Chaudron’s “convex simulation” and the new notion of 
“data-equivalent” simulation. 1

1 Introduction

1.1 The Gamma Model

The unique data structure of Gamma is a multiset of data items, called a state. 
A Gamma program consists of a set of atomic programs. These are rewrite rules 
(also called reaction rules) r of the form

\[ r : \text{lhs} \rightarrow f(\text{lhs}), \text{if cond(lhs)} \]

which states that whenever elements matching lhs in the multiset satisfy condition cond(lhs), they may be replaced in the multiset by the result of applying 
the function f to lhs. A result (not “the” result!) is obtained when a stable state 
is reached, i.e., when no more reaction can take place. For example the Gamma 
program consisting of just the following rule max computes the maximum ele-
ment of a non-empty multiset, whereas the following add sums up the elements 
of a non-empty multiset:

\[ \text{max} : \quad x, y \rightarrow y, \text{if } x \leq y \]

\[ \text{add} : m, n \rightarrow m + n, \text{if true} \]

1 A longer version of this paper, with detailed proofs, appears as a technical report 
and can be reached via (http://www.cs.chalmers.se/~martinw/papers/).
The semantics of an atomic Gamma rule $r$ is thus determined by a binary relation $\mathcal{R}_r$ between states: $(M, M') \in \mathcal{R}_r$ iff state $M$ can be changed into $M'$ by one application of rule $r$.

Atomic Gamma programs can be combined into bigger ones with sequential and parallel composition [6], which we write $p; q$ and $p \parallel q$.

The semantics of a composed Gamma program $p$ is given in form of an unlabelled transition system $\rightarrow$ on pairs of a program and a state, and a termination predicate $\sqrt{\cdot}$ ("immediate convergence") on such pairs. The semantics of $p$ is derived from the semantics $\mathcal{R}$ of its atomic programs by the set of Structured Operational Semantics rules:

$$
\frac{(M, M') \in \mathcal{R}_r}{(r, M) \rightarrow (r, M')} \quad \text{(at$_{\rightarrow}$)}
$$

$$
\frac{(p, M) \rightarrow (p', M')}{(p; q, M) \rightarrow (p'; q, M')} \quad \text{(Seq$_{\rightarrow}$)}
$$

$$
\frac{(p, M) \rightarrow (p', M')}{(p \parallel q, M) \rightarrow (p', \parallel q, M')} \quad \text{(Par$_{\rightarrow}$)}
$$

Gamma is thus an instance of a composed reduction system [9], which are all systems with a sequential and a parallel composition that behave according to this set of rules, independently of the underlying data structure $\text{State}$ or the way the one-step relations $\mathcal{R}_r$ are defined.

### 1.2 Refinement Relations

Gamma programs are related to each other by different forms of refinement. A simple Gamma program can be seen as a very nondeterministic (pre-)algorithm. A refinement step can make the program more specific and more deterministic. A program may (but need not necessarily) be refined into a completely deterministic algorithm. If a program $p$ is a refinement of a program $q$, written $p \preceq q$, then $p$ is considered a correct implementation of $q$.

Sometimes however, as in the "pipelining transformation" of this paper, we are also interested in "refining" in the other direction: from a more deterministic $p$ to a more general $q$, maybe in order to refine $q$ into a different $q'$ later on.

There are a variety of possible ways to describe the "behaviour" of a transition system, which also give rise to different refinement relations.

One such description which is often relevant is that of input-output behaviour. For any given input, we are interested in which output(s) the system can produce, and consider two systems equivalent if for every given input they will produce the same (set of) output(s). Mathematically, the semantics of the system is a relation between inputs and outputs (or a function in the deterministic case). In the case of a transition system as we have here, with transitions between program-state pairs, we can consider a state $M$ as the input to a program $p$ and consider all the transition sequences that start with $(p, M)$. If such