From Gamma to CBS: Refining multiset transformations with broadcasting processes

David Sands and Martin Weichert
Department of Computing Science
Göteborg University and Chalmers University of Technology
412 96 Göteborg, Sweden
{dave,martinw}@cs.chalmers.se

Abstract

This paper presents a novel approach to the problem of implementing programs in Gamma, a computation model of chemical-reaction-like multiset transformations, by translating them into a process calculus with broadcasting communication, CBS. The concurrent message reception of broadcasting communication fits very naturally to the implicit parallelism of the Gamma model: A value that may trigger reactions with several others in the multiset is broadcast to the potential receivers and may thus react with all of them at the same time. This kind of triggering reactions, which we call quasi-unary*, is very common in Gamma programs and is found in a large class of problems. The translation constitutes a correct refinement of the Gamma program and offers possibilities for further optimisations for several classes of problems. We address termination of Gamma programs and identify several classes of programs where termination can be detected and practically implemented.*[1]

1 Introduction

This paper explores some connections between Gamma and CBS. Gamma and CBS are two abstract programming paradigms which embody two key coordination mechanisms.

- The Gamma model is an abstract computation model based on the chemical reaction metaphor [BCL88, BL90, HLS92]. In the context of coordination mechanisms, Gamma serves as an abstract model in which the essence of generative communication in an unstructured data-space can be studied. It provides a high level specification language for algorithms, and also a mechanism for specifying concurrent semantics of programming languages, as popularised in the "Chemical Abstract Machine" [BB92].
- Calculus of Broadcasting Systems (CBS) [Pra95], a simple CCS-like calculus with an intuitive communication model, single channel broadcast. Processes speak one at a time and are heard instantaneously by all other processes. Speech is autonomous, contention being resolved non-deterministically. Hearing only happens when someone else speaks.

Although a number of papers have, in passing, pointed to similarities and possible connections between CBS and Gamma (and between CBS and other coordination languages, such as the broadcast-based language LO [AP90, ALPT93]), a more thorough investigation has not been forthcoming. A connection between broadcasting and the Gamma-inspired chemical abstract machine has been forgred in the development of *interaction abstract machines* [ACP93], which extend the chemical reaction metaphor with a notion of broadcast communication.

This paper describes how a certain class of Gamma programs can be realised very naturally in the CBS paradigm. In this setting, broadcasting offers a novel approach to the combinatorial problem of searching for applicable reactions — the fundamental problem when implementing Gamma programs. Informally, the class of programs that can be realised in CBS are those in which at most one element is changed during any reaction step. We call this kind of reaction *quasi-unary*. This turns out

[1] See *http://www.cs.chalmers.se/∼martinw/papers/hicss98.ps* for full version of this paper.

Copyright 1998 IEEE. Published in the Proceedings of the Hawai'i International Conference On System Sciences, January 6-9, 1997, Kona, Hawaii.

Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions/IEEE Service Center/445 Hoes Lane/P.O. Box 1331/Piscataway, NJ 08855-1331, USA. Telephone: + Intl. 908-562-3966.

1060-3425/98 $10.00 (c) 1998 IEEE

to be a very common case for Gamma programs, and example Gamma algorithms in this class found in the literature include, for example, *knapsack*, *shortest-path*, *longest upsequence*, *primes*, and *matrix multiplication*. For this class, the problem of implementing the implicit search for applicable reactions is realised by broadcast. This should provide a natural highly concurrent refinement of the possible Gamma execution. Although we focus on non-reactive programs, quasi-unary reactions are also common in reactive Gamma programs – for example all the (higher-order) reactions of the reactive system modelled in [Bou96] are quasi-unary.

Implementing the Gamma model There have been a number of concrete experiments in implementing the Gamma model. For example on

- MIMD machines in the original work of Banâtre et al. [BCL88], and by Kuchen and Gladitz [KG93, GK96];
- the connection machine [Cre91], and even
- Field-programmable gate arrays (FPGA's) [Vie96].

Each of these articles accepts that efficient implementations cannot be realised in general, and that algorithm-specific optimisations must be made to handle the combinatorial problems of searching for applicable reactions and of termination detection; however no systematic approaches to this problem are described.

Some recent approaches to the problems of implementing the general Gamma model seek to impose a coordination layer on Gamma programs so that they can be further refined in order to obtain reasonable implementations. Chaudron and de Jong [CdJ96] describe a process-calculus-like language for refining the order of reactions occurring in a Gamma program. Fradet and Le Métayer [FM96] focus on the organisation of the data itself, so that structure and locality can be programmed more naturally.

2 Introduction to the Gamma model

Gamma [BL90, BL93, HLS92] is a language based on local multiset rewriting with an elegant chemical reaction metaphor: A program state is a multiset of small data units, which — in analogy to molecules in a chemical solution — may interact locally according to a set of simple reaction rules. This simple programming model aims not to impose any sequential ordering on the computation other than those due to the problem itself. (For example, many programming languages force a list or an array to be traversed in a particular order even if the problem statement itself does not require that.) This openness of order of execution may be used to first specify a problem in a quite general solution in Gamma and then refine it into a more specific solution with a specific sequential — or even parallel — execution of the computation steps.

The unique data structure of Gamma is a multiset of data items, for example numbers or tuples, which we will call *cells*. The behaviour of the data structure is described by a set of *rewrite rules* (also called *reaction rules*). For the purposes of this article we will consider a Gamma program (ranged over by Γ) to be simply a collection of such rewrite rules. A rewrite rule of the form

$$lhs \rightarrow rhs, \text{ if } cond(lhs)$$

states that whenever elements matching *lhs* in the multiset satisfy condition *cond(lhs)*, they may be replaced in the multiset by the corresponding *rhs*.

If such a rewrite rule is in a program Γ and is applicable in a multiset M, resulting in a multiset N, then we will write $M \rightarrow_\Gamma N$.

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 following Gamma program computes the maximum element of a non-empty (multi-)set:

$$a, b \rightarrow a, \text{ if } a \geq b$$

Whenever two elements a and b in the multiset satisfy the condition $a \geq b$, they may be replaced by the single element a, that is, the lesser element b simply disappears. It is easy to see that reactions terminate iff the original multiset has reduced to the singleton multiset containing the maximum element (assuming a total ordering relation).

Assuming a certain multiset representation of the input and output data, many basic problems can be solved in Gamma with one single rewrite rule. Here are some more of them:
Eliminating duplicates:

$$a, b \rightarrow a, \text{ if } a = b$$

Adding up all elements:

$$m, n \rightarrow m + n, \text{ if } true$$

Finding prime numbers up to n:

$$m, n \rightarrow m, \text{ if } n \text{ is multiple of } m$$

applied to the start set $\{2, \ldots, n\}$.

Some more interesting examples:
Sorting: To represent a sequence $a_1, \ldots, a_n$ in Gamma, we will encode it as a multiset of pairs

1060-3425/98 $10.00 (c) 1998 IEEE

$\{(a_1,1),\ldots,(a_n,n)\}$ of a value and its index in the sequence. Two ill-ordered values will "exchange place" by exchanging their index:

$$(a,i),(b,j) \longrightarrow (a,j),(b,i), \text{ if } a < b \text{ and } i > j$$

Greatest common denominator (GCD) of a multiset of numbers:

$$a,b \longrightarrow a,b-a, \text{ if } 1 \le a \le b \qquad (1)$$

An additional rule "$a \rightarrow$, if $a = 0$" eliminates cells that become 0.

3 Introduction to CBS

Processes in CBS [Pra93a, Pra93b, Pra95, Wei95] are built from the basic operations $v!$, "output (broadcast) the value v", and $S?$, "input any value v from the set S". The operations used for translating Gamma programs are described as follows:

- $\mathbf{o}$, called the empty (or "dead") process, does nothing.
- The process $v! \cdot P$ can broadcast the value v and then behave like process P. Thus, $3! \cdot \mathbf{o}$ transmits the value 3 and then "dies".
- The process $\{x \mid c(x)\}? \cdot P(x)$ is able to receive any broadcast value v which satisfies the boolean condition $c(v)$ and then behave like process $P(v)$. It will (implicitly) ignore any broadcast v that does not satisfy $c(v)$. For example, $\{x \mid odd(x)\}? \cdot x! \cdot \mathbf{o}$ is able to do $v?$ (input value v) for any odd number v, and react to it with an output $v!$, after which it dies. Any even number is simply ignored by this process.
- The process $P_1 + P_2$ behaves either as P_1 or as P_2. It may start with any action (input or output) that P_1 or P_2 starts with; this first action determines whether it will continue as P_1 or as P_2.
- $P_1 \mid P_2$ means "P_1 and P_2 acting in parallel". In any parallel composition of two or more processes any process that is ready to speak may be picked (nondeterministically) to do so. All the other processes in parallel hear this at the same time and react accordingly: A process that is able to receive the value broadcast must do so; any process that is not able to receive it just remains unchanged.
- The process **if** c **then** P **fi** behaves like P if the boolean condition c holds, and does nothing otherwise. We also introduce the notation **if** c **then** P_1 **else** P_2 **fi** $=$ **if** c **then** P_1 **fi** $+$ **if** $\neg c$ **then** P_2 **fi**.

In $\{x \mid c(x)\}? \cdot P(x)$, x is *bound* in both c and P. The operator $\cdot$ binds stronger than $|$, and $+$ binds weakest. Some additional syntactical sugar will occasionally be used without further comment. Its meaning should in any case be clear from context.

The meaning of CBS processes is given in form of *labelled transitions*. (The precise operational semantics is given in the appendix, in the full version of the paper.) The transition $P \xrightarrow{v!} P'$ means that the broadcast value v emanates from the process P, after which the system behaves as P'. The transition $P \xrightarrow{v?} P'$ means that P hears a broadcast value v, and subsequently behaves as P'. For example, the behaviour of the odd number process above is expressed as:

$$\{x \mid odd(x)\}? \cdot x! \cdot \mathbf{o} \xrightarrow{v?} v! \cdot \mathbf{o} \xrightarrow{v!} \mathbf{o}$$

for any odd number v.

CBS processes are often defined recursively. A process definition like $P = \{x \mid odd(x)\}? \cdot x! \cdot P$ thus describes a process that echoes any odd number it hears and then returns to its initial state — it will thus repeatedly echo any odd number it hears:

$$P \xrightarrow{v?} v! \cdot P \xrightarrow{v!} P \xrightarrow{w?} w! \cdot P \xrightarrow{w!} P \xrightarrow{u?} \ldots$$

and so on for any odd numbers v, w, ... We write $P \rightarrow P'$ to mean that there exists some v for which $P \xrightarrow{v!} P'$.

Parallel composition is commutative and associative. It can thus be seen as an operation on a *multiset* of processes and we can write $\prod_{a \in M} P(a) = P(a_1) \mid \ldots \mid P(a_m)$ for any finite multiset $M = \{a_1,\ldots,a_m\}$.

Equal processes. Two processes are considered equal if they have the same behaviour, that is, if they have the same set of transitions, which again lead to equal processes. In process calculus, this is formally called *bisimulation*. We write $P \equiv Q$ or simply $P = Q$ for equal (that is, bisimilar) processes P and Q.

4 From Gamma to CBS

Gamma reactions may be thought of as being executed in parallel. Whenever disjoint subsets satisfy some reaction condition, like $a,b \rightarrow c$ and $d,e \rightarrow f$, we can think of one reaction $a,b,d,e \rightarrow c,f$ combining the two.

The essential idea in our approach is that with broadcasting and quasi-unary reactions we do not combine reactions on *disjoint* sets (CBS uses an interleaving semantics), but rather reactions that all share the same

1060-3425/98 $10.00 (c) 1998 IEEE

catalyst. Reactions $a, b \rightarrow a, b'$ and $a, c \rightarrow a, c'$ can be combined into $a, b, c \rightarrow a, b', c'$.[2]

4.1 Quasi-unary reactions

We call a rewrite rule *quasi-unary* (or *catalysing*) if it is of the form:

$$a, b \longrightarrow a, b'_1, \ldots, b'_m, \text{ if } cond(a, b)$$

In a quasi-unary reaction, exactly two cells a and b meet. Whereas one of them, a, remains unchanged, the other may be replaced by zero, one or more other cells. We say that a "catalyses" the reaction on b and write

$$b \xrightarrow{a} b'_1, \ldots, b'_m, \text{ if } cond(a, b) \qquad (2)$$

to emphasise that a is needed for the reaction but does not change itself.

Of all the examples above, the summing and the sorting programs are **not** quasi-unary, since two reactants change at the same time. The GCD reaction (1) however is, and we can write:

$$b \xrightarrow{a} b - a, \text{ if } 1 \leq a \leq b \qquad (3)$$

In section 6.1 we will meet a quasi-unary solution to sorting as well.

4.2 Translating quasi-unary reactions

Now we can translate an arbitrary quasi-unary rule (2) to the following process definition:

$$\begin{array}{rcl} P(b) & = & b! \cdot P(b) \\ & + & \{a \mid cond(a, b)\}? \cdot \left(P(b'_1) \mid \ldots \mid P(b'_m)\right) \end{array}$$

Any cell is always ready to announce its own value. It will not change by doing so. It is also always ready to hear a value that may catalyse it to change into something different.

The above translation of a single rule extends to an arbitrary set of quasi-unary and unary reactions, by incorporating a summand in the process description corresponding to each possible reaction condition. For unary reactions the process can modify its value autonomously. For notational simplicity we will stick to the simple case of a Gamma program consisting of a single reaction rule.

Executing the Gamma program on the multiset $\{a_1, \ldots, a_n\}$ can now be implemented (possibly reducing the nondeterminism) by the parallel composition $P(a_1) \mid \ldots \mid P(a_n)$. The GCD example (3) becomes:

$$\begin{array}{rcl} P(b) & = & b! \cdot P(b) \\ & + & \{a \mid 1 \leq a \leq b\}? \cdot P(b - a) \end{array}$$

[2]Both [CdJ96] and [CGZ96] extend the formal operational semantics of Gamma to include this kind of transition rule.

4.3 Correspondence between Gamma and CBS

We denote the translation of a Gamma program Γ by $\widehat{\Gamma}$, which is a mapping Elements $\rightarrow$ CBS. We extend this pointwise to a function on multisets of elements in the obvious way, so that $(\lambda x.P(x))M = \prod_{a \in M} P(a)$.

The translation yields processes which have a degree of evolving structure — reflecting the possible addition or elimination of elements from the multiset in the original Gamma program. This evolving structure is very regular in the sense that every transition has the form $\widehat{\Gamma}(M) \rightarrow \widehat{\Gamma}(N)$. Stated more precisely:

Lemma 1 *For all Gamma programs Γ, if $\widehat{\Gamma}(M) \rightarrow P$ then $P \equiv \widehat{\Gamma}(N)$ for some N.*

The first correspondence result states that these transitions can be simulated by the original Gamma program:

Proposition 1 (Simulation) *For every multiset M, and Gamma program Γ: If $\widehat{\Gamma}(M) \rightarrow \widehat{\Gamma}(N)$ then $M \rightarrow^*_\Gamma N$*

What is more, termination of the Gamma program implies stability of the CBS term:

Proposition 2 (Stability) *We say that a CBS term P is broadcast-stable if whenever $P \rightarrow Q$ then $P \equiv Q$. For all M, if $M \not\rightarrow_\Gamma$ then $\widehat{\Gamma}(M)$ is broadcast-stable.*

Liveness A final property that one might hope for is a liveness condition which states that the CBS translation will make progress whenever progress is possible. However, there is a hidden subtlety in the operational semantics of Gamma which any lower-level implementation must address: the transition rules for Gamma only reflect the successful reactions, and not the search process, which inevitably must test the reaction condition on many unsuccessful tuples. Corresponding to this, in the translated program it is possible to repeatedly broadcast a value which cannot react further with any other values in the system. One way around this problem is to assume fairness in the CBS implementation. The notion of fairness that is needed, informally, says that a process which is continuously able to broadcast a given value will eventually do so. This guarantees that every process which is able to broadcast a catalyst will eventually be able to do so. For completeness we give the formal statement below.

Proposition 3 *Under the above fairness assumption about CBS executions, if $M \rightarrow_\Gamma N$ for some $M \neq N$, then there exists an M' such that $\widehat{\Gamma}(M) \rightarrow^* \widehat{\Gamma}(M')$ and $N \rightarrow^*_\Gamma M'$.*

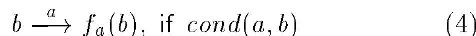
1060-3425/98 $10.00 (c) 1998 IEEE

5 Speeding it up: One broadcast does it all

In this section we consider a further optimisation which allows us to reduce unnecessary broadcasts. First we introduce a refinement of quasi-unary Gamma programs which reduces the number of reactions by iterating the reaction function until the reaction condition is false. For these programs we consider two related optimisations of the corresponding CBS programs which allow us to do a once-only ("one-shot") broadcast. An added bonus of these optimisation is that they guarantee progress without the need for fairness.

Speed-up by broadcasting How many times is a catalyst value needed? In Gamma, there is in principle no limitation on how many times a cell might react with others – with or without changing its own value. There are however some general considerations that will help us to limit, and even cut down on, the number of reactions a particular cell engages in.

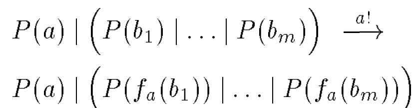
For simplicity we will consider a quasi-unary reaction of the form



$$b \xrightarrow{a} f_a(b), \text{ if } cond(a,b) \quad (4)$$

i.e., where the reaction does not change the size of the multiset. The right hand side is here written as $f_a(b)$, a function of both a and b. (As a function depending on two arguments we could equivalently write it as $f(a,b)$, but the notation $f_a(b)$ will be more convenient for our purpose later on.)

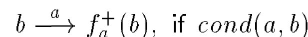
The translation from Gamma to CBS entails that there may be several cells $b_1, \ldots, b_m$ around that satisfy the reaction condition to react with catalyst a, that is, $cond(a, b_1) = \ldots = cond(a, b_m) = true$, and the multiset $a, b_1, \ldots, b_m$ may do several reaction steps in Gamma to become $a, f_a(b_1), \ldots, f_a(b_m)$. It is exactly this kind of reactions that fits well with broadcast communication, as all of these Gamma reactions will merge into one single transition in the process calculus case:



$$P(a) \mid \Big(P(b_1) \mid \ldots \mid P(b_m)\Big) \xrightarrow{a!}$$
$$P(a) \mid \Big(P(f_a(b_1)) \mid \ldots \mid P(f_a(b_m))\Big)$$

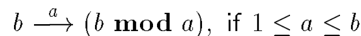
Thus broadcasting may reduce the number of steps by the order $O(n)$ where n is the number of cells.

Iterated reactions The fact that a catalyst is broadcast does not rule out the case where the *same* cell b can be catalysed by the same value a more than once, that is when the resulting value $f_a(b)$ again satisfies the reaction condition $cond(a, f_a(b)) = true$. In this case there may still be need for an iterated broadcast communication of a, to trigger the reactions $b \xrightarrow{a} f_a(b) \xrightarrow{a} f_a(f_a(b)) \longrightarrow \ldots$ An example of this is the greatest common denominator algorithm (3) with $cond(a,b) = 1 \leq a \leq b$ and $f_a(b) = b - a$, with reaction chains like $70 \xrightarrow{20} 50 \xrightarrow{20} 30 \xrightarrow{20} 10$. In such a case, it is actually not at all necessary for cell b to "be informed" about a's value again – it already "knows" after the first reaction that the catalysing value a exists in the environment and that it therefore can go all the way from b to $f_a(f_a(b))$ if $cond(a, f_a(b)) = true$, and so on. Actually, if we have a termination proof of the Gamma program, we can even assume that every transition goes *as far as possible* in applying f_a, by defining the following:



$$f_a^+(b) = f_a^m(b), \text{ where}$$
$$m = \min\{m \mid cond(a, f_a^m(b)) = \mathit{false}\}$$

and replacing the original reaction rule (4) by:



$$b \xrightarrow{a} f_a^+(b), \text{ if } cond(a,b)$$

Example: Iterated GCD For the GCD algorithm (3), the iterated reaction function has a simple closed-form: $f_a^+(b) = b \textbf{ mod } a$.

The improved Gamma program is:

$$b \xrightarrow{a} (b \textbf{ mod } a), \text{ if } 1 \leq a \leq b$$

In general, turning a quasi-unary reaction function f_a into the iterated reaction f_a^+ in a Gamma program is well defined and preserves the input-output behaviour on the program's termination domain. The termination domain of a Gamma program Γ, written $Dom(\Gamma)$ is the set of multisets for which Γ *must* terminate (i.e. cannot run forever). The termination domain is of relevance to programs which satisfy a given pre/postcondition specification: a totally correct Gamma program with respect to precondition p and postcondition q is such that whenever M satisfies p then we must have $M \in Dom(\Gamma)$. In the terminology of [HLS92], replacing f_a by f_a^+ is a program refinement in the upper (total-correctness) ordering.

One-shot announcements Once we have converted the Gamma reactions to iterated form, we now consider the question of reducing the number of broadcasts. We consider two types of optimisation:

1. **Value one-shot broadcast** In this optimisation a cell does not broadcast a given value more than once (unless the value changes); In this case we can optimise the reaction condition by augmenting each

1060-3425/98 $10.00 (c) 1998 IEEE

$$
\begin{array}{rcl}
P(b, new) & = & \textbf{if } new \textbf{ then } b! \cdot P(b, \mathit{false}) \textbf{ fi} \\
& + & \{a \mid cond(a,b)\}? \cdot \Big(P(b'_1, true) \mid \ldots \mid P(b'_m, true)\Big)
\end{array}
$$

Figure 1: CBS process definition for value one-shot broadcast

CBS process with an additional, boolean parameter new, initially set to $true$. The CBS process definition is given in Figure 1.

2. **Cell one-shot broadcast** A variant of the above in which a cell broadcasts at most one value. The CBS process definition is given in Figure 2.

Neither of these optimisations is sound in general. The necessary condition for them to be sound is, roughly speaking, that if the processes were to do repeated broadcasts, then these would not trigger any further reactions. In many Gamma programs, we have quasi-unary rules that are "one-shot". It is, however, not always obvious that this is the case. We will first look at some examples where these optimisations can be applied, before stating sufficient conditions on the condition and reaction function to enable these optimisations.

An Example Value One-Shot Reaction: GCD The reaction is value one-shot, since once we have broadcast c, the reaction ensures that all other elements will be smaller than c, and since the reaction never increases the size of the values, this reaction condition is never re-enabled. The process above becomes:

$$
\begin{array}{rcl}
P(b, new) & = & \textbf{if } new \textbf{ then } b! \cdot P(b, \mathit{false}) \textbf{ fi} \\
& + & \{a \mid 1 \leq a \leq b\}? \cdot P(b \textbf{ mod } a, true)
\end{array}
$$

Example Cell One-Shot Reaction: Transitive closure of a relation The transitive closure problem is, given a relation R on a (finite) set A, compute R^+, its transitive closure. We will first sketch the derivation of a Gamma program for this problem. To solve this in Gamma we can represent any relation R' as a (multi)set of cells, each containing one element of A and the set of its "right neighbours":

$$M(R') = \{(a, S_a) \mid a \in A, S_a = \{b \mid (a,b) \in R'\}\}$$

The initial condition I is: $M = M(R)$ and the final condition T is $M = M(R^+)$. A suitable invariant is $\exists R' : M = M(R') \wedge R \subseteq R' \subseteq R^+$. Reactions must still take place as long as $\neg T$ holds, together with the invariant, that is, when $R \subseteq R' \subset R^+$, i.e.,

$$\exists a, b, c : (a,b) \in R', (b,c) \in R', (a,c) \notin R'$$

This condition is equivalent to:

$$\exists (a, S_a), (b, S_b) : b \in S_a \wedge S_b \not\subseteq S_a$$

This gives a suitable reaction condition on the cells (a, S_a) and (b, S_b). As an appropriate reaction we can complete S_a with the missing elements of S_b:

$$(a, S_a), (b, S_b) \longrightarrow (a, S_a \cup S_b), (b, S_b), \text{ if } b \in S_a \wedge S_b \not\subseteq S_a$$

The reaction rule is quasi-unary:

$$(a, S_a) \xrightarrow{(b, S_b)} (a, S_a \cup S_b), \text{ if } b \in S_a \wedge S_b \not\subseteq S_a$$

and we can translate it to CBS.

$$
\begin{array}{rcl}
P(a, S_a) & = & (a, S_a)! \cdot P(a, S_a) \\
& + & \{(b, S_b) \mid b \in S_a \wedge S_b \not\subseteq S_a\}? \cdot P(a, S_a \cup S_b)
\end{array}
$$

It can be shown (using the sufficient condition given in section 5.1) that once the condition $b \in S_a \wedge S_b \not\subseteq S_a$ is false for a particular b, it can never become $true$ again by any subsequent reactions. The reaction rule is thus cell one-shot and we get:

$$
\begin{array}{l}
P(a, S_a, new) = \\
\quad \textbf{if } new \textbf{ then } (a, S_a)! \cdot P(a, S_a, \mathit{false}) \textbf{ fi} \\
\quad + \{(b, S_b) \mid b \in S_a \wedge S_b \not\subseteq S_a\}? \cdot P(a, S_a \cup S_b, new)
\end{array}
$$

This also gives us an easy termination proof, since $\sum_a |S_a|$ increases in every reaction step and is bound above by $|A|^2$.

The resulting CBS process can further be refined "manually". If, for example, it is felt inappropriate to communicate whole sets in single messages (which might be limited in size), each message $(a, \{b_1, \ldots, b_m\})!$ could be replaced by a sequence $(\mathsf{start}, a)! \cdot b_1! \cdot \ldots \cdot b_m! \cdot \mathsf{done}!$ of one-element messages.

5.1 Sufficient Conditions for One-Shot Optimisations

Here we state sufficient conditions for the respective one-shot optimised CBS programs. We consider the simple case of a reaction of the form

$$b \xrightarrow{a} f_a^+(b), \text{ if } cond(a,b)$$

Let f_a^* denote the function which is like f_a^+ on the domain of $cond(a, _)$, and behaves like the identity function elsewhere.

1060-3425/98 $10.00 (c) 1998 IEEE

$$
\begin{aligned}
P(b, new) &= \textbf{if } new \textbf{ then } b! \cdot P(b, false) \textbf{ fi} \\
&+ \{a \mid cond(a,b)\}? \cdot \Big(P(b'_1, new) \mid \ldots \mid P(b'_m, new)\Big)
\end{aligned}
$$

Figure 2: CBS process definition for cell one-shot broadcast

Proposition 2 *The value once-only optimisation is sound under the following condition:*

$$\forall c, d, e : \neg cond(c,d) \Rightarrow \neg cond(c, f_e^+(d))$$

The intuition is that once c has been broadcast, $\neg cond(c,d)$ holds for all d (since the reaction is iterated). Then the condition $\neg cond(c, f_e^+(d))$ ensures that c cannot react with any descendent of d.

Proposition 3 *The cell once-only optimisation is sound under the following condition:*

$$
\begin{aligned}
\forall c, d, e : \quad & \neg cond(c,d) \Rightarrow \\
& \neg cond(f_e^*(c), f_e^*(d)) \wedge \neg cond(f_d^*(c), d)
\end{aligned}
$$

In this case, once c has broadcast we have $\neg cond(c,d)$. We need to ensure that this holds for all descendent of both d *and* c. The immediate descendents of c and d arise through either a broadcast of a third element e, in which case the condition $\neg cond(f_e^*(c), f_e^*(d))$ guarantees that these cannot react, or d itself could be broadcast, explaining the second conjunct. The case where c is broadcast is not considered, since the premise ensures that this cannot change d.

The GCD reaction is easily seen to satisfy the condition of Proposition 2; it is routine to verify that the transitive-closure example satisfies the condition of Proposition 3.

6 Termination detection

Usually we want Gamma programs to terminate. One problem with termination is proving that a program eventually will terminate — this can suitably be done with a function from multisets to a well-founded ordering, where every reaction strictly decreases the value of that function (for Gamma examples see [BL90]).

Even if we know that a program eventually must reach a terminating state, we will still have to face the problem of detecting if this already has happened or if there are still more reactions possible.

[HLS92] studies a *sequential composition* operator between Gamma programs. The sequential composition $\Gamma_2 \circ \Gamma_1$ of two Gamma programs Γ_1 and Γ_2 is a program that first executes Γ_1, and then, if and when Γ_1 terminates, goes on with Γ_2, with the final state (multiset) of Γ_1 being the starting one for Γ_2. This is similar to sequential composition $P_1 \cdot P_2$ in process calculus. However the difficulty in Gamma lies in detecting the termination of Γ_1 — finding out when no more reactions are possible. In process calculus the first component P_1 is usually just a single process where we have no difficulty detecting the termination.

Let us consider a Gamma program Γ_1 consisting of a quasi-unary reaction:

$$b \xrightarrow{a} b'_1, \ldots, b'_m, \text{ if } cond(a,b)$$

We want to code this program into a CBS process definition $P(b)$ in such a way that it can be sequentially composed with another program Γ_2 which has been coded into $Q(b)$. That means that if and when no more reactions of Γ_1 are possible, all the cells must, immediately and synchronously, stop behaving according to Γ_1 and start behave according to Γ_2: Every cell $P(b)$ must turn into some cell $Q(b)$.

The termination condition in general is: "no more reactions are possible". For one binary (including quasi-unary) reaction, this is written formally as: $\forall a \in M : \forall b \in M : \neg cond(a,b)$.

In several programs, it can be shown, by some problem-specific properties, that the global termination condition is equivalent to some other, simpler condition. We can characterise several simplified termination conditions which often arise in practice:

1. "local-unary": A condition of the form: $\exists a \in M : done(a)$.
2. "global-unary": A condition of the form: $\forall a \in M : done(a)$.
3. "number of cells": A condition of the form: $|M| = 1$ (or some other constant).
4. "number of broadcasts": The program terminates if and when a constant number of broadcasts has occurred.
5. "number of reactions": Similar for the number of reactions.

For each kind of condition we will show how termination can be programmed in CBS.

1060-3425/98 $10.00 (c) 1998 IEEE

6.1 "local-unary"

A condition of the form: $\exists a \in M : done(a)$. Termination is detected locally by any one cell if it satisfies a condition on its stored value(s). With broadcast communication, the cell for which *done* becomes *true* can then transmit a signal done to all the other cells, after which all cells synchronously start with another program Q instead.

Whenever we have a quasi-unary reaction $b \xrightarrow{a} b'_1, \ldots, b'_m$ with a local-unary termination condition *done*, we can code this automatically into CBS:

$$
\begin{array}{l}
P(b) = \\
\quad \textbf{if } done(b) \textbf{ then } \mathsf{done}! \cdot Q(b) \\
\quad \textbf{else} \quad b! \cdot P(b) \\
\qquad\qquad + \{a \mid cond(a,b)\}? \cdot \big(P(b'_1) \mid \ldots \mid P(b'_m)\big) \\
\qquad\qquad + \{\mathsf{done}\}? \cdot Q(b) \\
\quad \textbf{fi}
\end{array}
$$

Example: Sorting We start with the multiset $\{(a_1, 1), \ldots, (a_n, 1)\}$ and use the following quasi-unary reaction:

$$(b, j) \xrightarrow{(a,j)} (b, j+1), \text{ if } a \le b$$

We can easily see that the following invariant properties hold:

1. $\{a \mid (a,j) \in M\} = \{a_1, \ldots, a_n\}$
2. $\exists (a,j) \in M : j = 1$
3. $\forall (b,j) \in M, j > 1 : \exists a \le b : (a, j-1) \in M$
4. $\forall (b,j) \in M, j > 1 : \exists a_1 \le \ldots \le a_j = b : (a_1, 1), \ldots, (a_j, j) \in M$
(by induction, using the previous property)

The system will terminate:

iff no more reactions are possible,

$\iff$ all cells have different rank (we assume the ordering relation is total),

$\iff$ the occurring ranks are exactly $\{1, \ldots, n\}$ (by the last invariant),

$\iff$ there exists exactly one cell with rank n.

Therefore we have the "local-unary" termination condition $\exists (a,j) \in M : done(a,j)$, where $done(a,j) = (j = n)$.

The reaction rule is "one-shot": after any reaction $(b,j) \xrightarrow{(a,j)} (b, j+1)$, the rank $j+1$ can only further increase and never come back down to j again. We apply the coding above and add an additional argument *new* for the one-shot property.

$$
\begin{array}{l}
P(b, j, new) = \\
\quad \textbf{if } j = n \textbf{ then } \mathsf{done}! \cdot Q(b,j) \\
\quad \textbf{else} \quad \textbf{if } new \textbf{ then } (b,j)! \cdot P(b, j, false) \textbf{ fi} \\
\qquad\qquad + \{(a,j) \mid a \le b\}? \cdot P(b, j+1, new) \\
\qquad\qquad + \{\mathsf{done}\}? \cdot Q(b,j) \\
\quad \textbf{fi}
\end{array}
$$

6.2 "global-unary"

A condition of the form: $\forall a \in M : done(a)$

Termination can be detected whenever **all** cells satisfy a certain condition on their stored values. We can use a global-unary termination condition if we have a Gamma rule $b \xrightarrow{a} f_a(b)$, if $cond(a,b)$ (i.e., the number of cells does not increase). The idea is to keep a counter of how many cells still need to reach the termination condition *done*. A cell that reaches *done* announces this publicly to decrement the counter. This counter could be kept in an extra cell, different from all the others. Instead, we choose here to let each cell keep a copy of the counter, which makes for a more uniform solution. Due to the broadcasting communication, we can let all cells decrease their counters in lock-step. A boolean parameter *told* keeps track on whether the cell has already announced itself.

$$
\begin{array}{l}
P(b, ctr, told) = \\
\quad \textbf{if } ctr = 0 \textbf{ then } Q(b) \\
\quad \textbf{else} \quad \textbf{if } done(b) \wedge \neg told \\
\qquad\qquad \textbf{then } \mathsf{done}! \cdot P(b, ctr-1, true) \textbf{ fi} \\
\qquad\qquad + b! \cdot P(b, ctr, told) \\
\qquad\qquad + \{a \mid cond(a,b)\}? \cdot P(b', ctr, told) \\
\qquad\qquad + \{\mathsf{done}\}? \cdot P(b, ctr-1, told) \\
\quad \textbf{fi}
\end{array}
$$

Even without such a unary termination condition *done*, but with the optimisation for value-one-shot reactions, we still have the general unary condition on every cell that it has (temporarily) nothing to say. Every cell with a *new* value will announce that value once and then fall (temporarily) passive, still listening to incoming broadcasts. A suitable incoming broadcast may reactivate the passive cell. If in such a system the number of cells remains constant $= n$, then it will terminate iff the number of "passive" cells is $= n$. A cell knows if it is passive ($new = false$), but it does not know in general how many of the others are. In order to find out, we can let any passive cell participate in a "roll count" at any time, by announcing done! once and counting the incoming done!'s. If, and only if, *all* cells are passive, the roll count will reach n for all of them, at the same time, and they can simultaneously change into something else. In any other case, the roll count cannot reach up to n; instead, a "real" broadcast will interfere, to which all the passive cells react by resetting their counters to zero (and

possibly by turning into active cells again, depending of the value of that broadcast). We therefore get:

$$
\begin{aligned}
&P(b, new, ctr, told) = \\
&\quad \textbf{if } ctr = n \textbf{ then } Q(b) \\
&\quad \textbf{else} \quad \textbf{if } new \\
&\qquad\qquad \textbf{then } b! \cdot P(b, false, 0, false) \textbf{ fi} \\
&\qquad\quad + \{a \mid cond(a,b)\}? \cdot P(f_a(b), true, 0, false) \\
&\qquad\quad + \{a \mid \neg cond(a,b)\}? \cdot P(b, new, 0, false) \\
&\qquad\quad + \textbf{if } \neg new \wedge \neg told \\
&\qquad\qquad \textbf{then } \mathsf{done}! \cdot P(b, false, ctr+1, true) \textbf{ fi} \\
&\qquad\quad + \textbf{if } \neg new \\
&\qquad\qquad \textbf{then } \{\mathsf{done}\}? \cdot P(b, false, ctr+1, told) \textbf{ fi} \\
&\quad \textbf{fi}
\end{aligned}
$$

This coding will guarantee termination detection for the value-one-shot reaction with constant number of cells. As it stands, it is far from being any efficient implementation, because of all the intervening done! broadcasts. In the worst case, every real broadcast will be followed by a number of done!'s; in the best case there will be only n done!'s at the very end. At the same time, this coding may be used as a template for problem-specific optimisations. In many cases the $\neg new$ condition can be augmented with an additional (problem-specific) condition on b indicating that a cell, even though it is passive itself, may *know* that there are non-passive cells around and that it therefore can abstain from participating in the "roll count". Such an optimisation can considerably cut down on the number of "unnecessary" done! messages.

6.3 "number of cells"

A similar case is the condition $|M| = k$, where the cells vanish until there are only k (usually one) of them left. (Here too it only works if the number of cells does not increase.) We can keep track with a counter in a similar way. For example, the GCD program reduces to exactly one cell if every cell that reaches 0 disappears. We use a counter argument ctr again, plus the boolean argument new for the one-shot property:

$$
\begin{aligned}
&P(b, new, ctr) = \\
&\quad \textbf{if } b = 0 \textbf{ then } \mathsf{done}! \cdot \mathbf{o} \\
&\quad \textbf{elsif } ctr = 1 \textbf{ then } Q(b) \\
&\quad \textbf{else} \quad \textbf{if } new \textbf{ then } b! \cdot P(b, false, ctr) \textbf{ fi} \\
&\qquad\quad + \{a \mid a \le b\}? \cdot P(b \textbf{ mod } a, true, ctr) \\
&\qquad\quad + \{\mathsf{done}\}? \cdot P(b, new, ctr-1) \\
&\quad \textbf{fi}
\end{aligned}
$$

As another example, finding the maximum of a multiset:

$$
\begin{aligned}
&P(b, new, ctr) = \\
&\quad \textbf{if } ctr = 1 \textbf{ then } Q(b) \\
&\quad \textbf{else} \quad \textbf{if } new \textbf{ then } b! \cdot P(b, false, ctr) \textbf{ fi} \\
&\qquad\quad + \{a \mid a \ge b\}? \cdot \mathsf{done}! \cdot \mathbf{o} \\
&\qquad\quad + \{\mathsf{done}\}? \cdot P(b, new, ctr-1) \\
&\quad \textbf{fi}
\end{aligned}
$$

6.4 "number of broadcasts"

In some algorithms it may be the number of *broadcasts* that is constant, regardless of the sequence in which they take place. In this case, if every cell keeps counting down, we do not need any additional communication. This program will terminate after exactly n broadcasts, if all the cells are initialised to $ctr = n$:

$$
\begin{aligned}
&P(b, ctr) = \\
&\quad \textbf{if } ctr = 1 \textbf{ then } Q(b) \\
&\quad \textbf{else} \quad b! \cdot P(b, ctr-1) \\
&\qquad\quad + \{a \mid \neg cond(a,b)\}? \cdot P(b, ctr-1) \\
&\qquad\quad + \{a \mid cond(a,b)\}? \\
&\qquad\qquad \cdot \Big(P(b'_1, ctr-1) \mid \ldots \mid P(b'_m, ctr-1)\Big) \\
&\quad \textbf{fi}
\end{aligned}
$$

This will usually be combined with the one-shot optimisation.

An example for this termination condition is the program for transitive closure, where we have exactly n cells, and each of them will broadcast exactly once.

6.5 "number of reactions"

Similarly, it could be the number of *reactions* which could be constant. For example we can predict that the sorting algorithm terminates after exactly $\frac{n \cdot (n-1)}{2}$ reactions, though we cannot know the number of *broadcasts* it needs. In such a case, we can equip each cell with a counter which is initialised to the number of expected reactions and run the following process:

$$
\begin{aligned}
&P(b, ctr) = \\
&\quad \textbf{if } ctr = 0 \textbf{ then } Q(b) \\
&\quad \textbf{else} \quad b! \cdot P(b, ctr) \\
&\qquad\quad + \{\mathsf{done}\}? \cdot P(b, ctr-1) \\
&\qquad\quad + \{a \mid cond(a,b)\}? \\
&\qquad\qquad \cdot \Big((\mathsf{done}! \cdot \mathbf{o}) \mid P(b'_1, ctr) \mid \ldots \mid P(b'_m, ctr)\Big) \\
&\quad \textbf{fi}
\end{aligned}
$$

This can of course also be combined with the one-shot optimisations.

7 Conclusion

We have studied the multiset programming model Gamma and presented a translation from an important class of Gamma programs to a process calculus with

1060-3425/98 $10.00 (c) 1998 IEEE

broadcast communication. The translation faithfully simulates reaction in Gamma and can therefore be seen as a correct refinement of the original program. At the same time it shows how broadcast communication, an otherwise often neglected paradigm of communication, is particularly suited to exploit the inherent parallelism of Gamma programs, without imposing too much unnecessary sequentiality. We addressed the problematic question of termination of Gamma programs and identified several classes of programs where a suitable invariant can guarantee termination detection. These Gamma programs can then mechanically be translated to broadcasting processes. There exists a simulator [Wei95], written in Haskell, for such processes, in which they then can be run as programs.

References

[ACP93] Jean-Marc Andreoli, Paolo Ciancarini, and Remo Pareschi. Interaction abstract machines. In G. A. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*, pages 257–280. MIT Press, 1993.

[ALPT93] JM. Andreoli, L. Leth, R. Pareschi, and B. Thomsen. True Concurrency Semantics for a Linear Logic Programming Language with Broadcast Communication. In *Proc. Conf. on Theory and Practice of Software Development (TAPSOFT 93)*, volume 668 of *LNCS*, pages 182–198, France, 1993. Springer.

[AP90] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*. MIT Press, 1990.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *TCS*, 96(217-248), 1992.

[BCL88] J-P. Banâtre, A. Coutant, and D. Le Métayer. A parallel machine for multiset transformation and its programming style. In *Future Generation Computer Systems*, volume 4, pages 133–144. 1988.

[Bes93] Eike Best, editor. *Proceedings CONCUR'93*, volume 715 of *LNCS*, Hildesheim, August 1993. Springer Verlag.

[BL90] J.-P. Banâtre and D. Le Métayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[BL93] J.-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *CACM*, 36(1):98–111, January 1993. (INRIA research report 1205, April 1990).

[BM92] J.-P. Banâtre and D. Le Métayer, editors. *Research Directions in High-level Parallel Programming Languages*. Springer-Verlag, LNCS 574, 1992.

[Bou96] M. Bourgois. Advantages of formal specifications: a case study of replication in Lotus Notes. In *Proc. Conference on Formal Models in Open Object-based Distributed Systems*, LNCS, Paris, March 1996. Springer-Verlag.

[CdJ96] M. Chaudron and E. de Jong. Towards a compositional method for coordinating Gamma programs. In P. Ciancarini and C. Hankin, editors, *Coordination'96 Conference*, volume 1061 of *Lecture Notes in Computer Science*, pages 107–123. Springer-Verlag, 1996.

[CGZ96] P. Ciancarini, R. Gorrieri, and G. Zavattaro. An alternative semantics for the parallel operator of the calculus of Gamma programs. In J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors, *Coordination programming: mechanisms, models and semantics*. IC Press, 1996.

[Cre91] C. Creveuil. Implementation of gamma on the connection machine. In *[BM92]*, 1991.

[FM96] P. Fradet and D. Le Métayer. Structured Gamma. Technical Report 989, IRISA, 1996.

[GK96] K. Gladitz and H. Kuchen. Shared memory implementation of the gamma-operation. *Journal of Symbolic Computation*, 21(4-6):577–591, April–June 1996.

[HLS92] C. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. Research Report DOC 92/22, Department of Computing, Imperial College, 1992.

[KG93] Herbert Kuchen and Katia Gladitz. Parallel implementation of bags. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 299–307, New York, NY, USA, June 1993. ACM Press.

[Pra93a] K. V. S. Prasad. A calculus of value broadcasts. In *PARLE'93*, volume 694 of *LNCS*. Springer Verlag, June 1993.

[Pra93b] K. V. S. Prasad. Programming with broadcasts. In *[Bes93]*, pages 173–187, 1993.

[Pra95] K. V. S. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25:285–327, 1995.

[Vie96] M. Vieillot. *Mise en oeuvre de l'opérateur Gamma à l'aide de circuits logiques reconfigurable*. PhD thesis, Irisa, Rennes, 1996.

[Wei95] Martin Weichert. FCBS: A forked calculus of broadcasting systems. Licentiate thesis, University of Göteborg and Chalmers University of Technology, November 1995.