Advantages of Formal Specifications: A Case Study of Replication in Lotus Notes

Marc Bourgois (marc@ecrc.de)

European Computer-Industry Research Center, Munich

Abstract: We show the advantages of formal specifications for distributed systems. We avoid contrived toy examples. Instead we focus on object replication in Lotus Notes, a popular groupware product, for which an informal specification already exists.

We base our formal specification on higher-order multiset rewriting. Using this formalism, we detect an error in the original specification, eliminate redundant and overly restrictive data structures, and expose the natural parallelism of the replication algorithm.

As a result we end up with a specification of the replication algorithm which is "more correct", leaner, and more widely usable. Indeed, the scope of the algorithm now includes parallel implementations, which may conceivably serve as the basis for highly available object servers on the Internet.

Keywords: formal specifications, multiset rewriting, distributed algorithms, Lotus Notes, refinement and transformation of specifications.

1. Introduction

We are concerned with formal methods as specification tools. Concretely we contrast specifications based on higher-order multiset rewrite rules with informal specifications. Higher-order multiset rewrite rules show their particular strengths in the specification of three aspects of modern software:

- *distribution:* Multiset rewriting is highly parallel in nature, thus capturing one of the most distinctive features of distributed systems: their parallelism [1].

- *object-based:* The higher-order extension of multiset rewriting provides a straightforward means of modeling the hierarchical, compositional aspects of object-based systems.
- *openness:* Higher-orderness also allows us to explicitly manipulate the rules, and thus (part of) the specification itself. This provides a means of modularizing, reusing, and exchanging parts of a specification. Such specifications become *open*, in the sense of being amenable, or easily adaptable, or reflective.

For a convincing proof of the relevance of our approach we selected an existing, commercially important and sizeable piece of software: Lotus Notes. Apart from being the most widely adopted groupware system, the Lotus Notes architecture and algorithms are well documented, making it an attractive case study.

In a previous report we have already used higher-order multiset rewriting to specify the distributed run-time system of a coordination language [3]. We were especially successful in exposing the programmable, or reflective, components of the run-time system. In this report we will mainly concentrate on the distribution and object-based aspects of multiset rewrite specifications.

Overview

The next section discusses the Lotus Notes system, and in particular its object replication algorithm. In section 3 we give an overview of higher-order multiset rewriting, the formalism used. Thereafter we develop a formal specification for the algorithm, closely matching the stepwise informal specification. Section 5 concentrates on the advantages of the formal specification: correctness, conciseness, and concurrency. We close with a critical discussion of the lessons learned.

2. Lotus Notes and its Replication Algorithm

Lotus Notes supports groups of people working on shared sets of documents. Documents are semi-structured objects which intermix graphs, images, pictures and numerical information with text. Sets of documents form databases.

Rather than locating the databases on a central server, the Notes system opted for a distributed architecture in which each participant works on local replicas of the shared databases. There is no master replica of a database. Propagation of database changes occur between pairs of replicas as a background activity. Conflicts are naively resolved on the basis of timestamps.

The replication algorithm guarantees "ultimate consistency" for applications where changes to existing documents are relatively infrequent compared to additions of new documents, and where small propagation delays do not matter. Asynchronous conferencing systems, such as bulletin boards, are typical examples. The phenomenal success of Lotus Notes proves that many groupware applications can easily do without the strict consistency and transaction capabilities traditional database management systems excel in.

Concretely, the Lotus Notes replication algorithm is a one-way pull model. It is so named because the algorithm is executed on the local computer and only pulls newer versions of documents from a remote computer. The algorithm consists of three steps:

1. *Create a list of databases (requiring replication)*
2. *Create a list of documents (requiring replication)*
3. *Replicate listed documents*

Though originally conceived for database replication among groups of geographically dispersed and infrequently connected participants, the algorithm has also been used for static load balancing and automatic backups. The backup strategy exploits the property that if one remote replica is unavailable, any other replica can be selected; which demonstrates the robustness of the algorithm.

Of course there is more to Lotus Notes than object replication. Other crucial features such as access control and locking fall outside the scope of this discussion. Partly for reasons of brevity, but partly also because the available specification is far less complete and explicit.

3. Higher-Order Multiset Rewriting

We now give a short overview of higher-order multiset rewriting. The language we use is akin to Higher-Order Gamma, for which a Structural Operational Semantics is given in [6].

The first concept we introduce is the *multiset*. A multiset is a container for elements, but unlike the *set* concept it may contain multiple copies of its elements. The empty set is also a multiset. The elements of multisets belong to basic types like timestamp, identifiers, strings or integers.

$$multiset \equiv \varnothing \,|\, \{value_1, \ldots, value_n\}.$$
$$value \equiv t \,|\, id \,|\, string \,|\, integer \,| \ldots$$

The multiset is the least restrictive type of container: It does not impose an order, or any other structure, upon its elements.

We use the Chemical Reaction Metaphor [2] to illustrate the different concepts in this section. In this analogy a chemical solution (something like a soup of molecules) represents the multiset, and molecules represent the elements of the multiset.

Rules

Another concept central to multiset rewriting is the *rule*. A rule consists of two parts: the *rewrite* and the *condition*. The rewrite in turn consists of two further parts: the head, or *reaction*, and the body, or *action*. Empty bodies or conditions are generally omitted for the sake of conciseness and readability.

$$rule \equiv (reaction \rightarrow action) \leftarrow condition$$

Let us continue with the metaphor. For a chemical reaction to take place, external factors such as temperature and pressure must be within specified ranges. Analogous restrictions are specified by a boolean expression in the condition part of the rewrite rule.

If all molecules in the head of the formula are present in the chemical solution, a reaction can take place. As a result the reaction molecules will be replaced in the solution by the action molecules. Triggering a chemical reaction corresponds to matching the patterns of the rewrite rule against the multiset. Patterns consist of grounded values or simple variables, matching any value.

$$\begin{aligned} reaction &\equiv pattern_1, \ldots, pattern_n \\ action &\equiv pattern_1, \ldots, pattern_n \end{aligned}$$

A chemical reaction can occur multiple times, potentially simultaneously, if the necessary molecules are multiply present in the chemical solution. The metaphor thus intuitively clarifies a crucial property of multiset rewriting: the inherent, massive parallelism.

Programs

The simplest programs are individual rules. We can construct more complex programs by using the parallel and sequential program combinators + and *o* [4]. Note that the sequential composition operator associates to the left.

$$progr \equiv rule \mid (progr_1 + progr_2) \mid (progr_2 \circ progr_1)$$

The notion of program composition is crucially dependent on the notion of program termination: A simple program executes for as long as its rule is applicable. Or, in our analogy, a chemical reaction continues for as long as sufficient reaction molecules are present in the solution.

A parallel program terminates when each constituent program terminates. A sequential program terminates when its last (i.e left-most) constituent program terminates.

Configurations

The language, as presented in the previous subsections, has little support for modularity. Two higher-order extensions solve this problem. The first extension allows us to explicitly represent multisets within the language. Consequently we can have multiple multisets and we can include multisets as elements of other multisets.

$$value \equiv t \mid id \mid string \mid integer \mid \ldots \mid multiset \mid configuration$$

The second extension allows us to explicitly represent programs within the language. Consequently we can bind a program to a set of explicitly named multisets. We call the resulting data structures active configurations.

$$\begin{aligned} configuration &\equiv passive \mid active \\ passive &\equiv (name_1 = value_1, \ldots, name_n = value_n) \\ active &\equiv [progr, passive] \end{aligned}$$

Note: A passive configuration is equivalent to an active configuration with an empty program.

$$[\varnothing, passive] \equiv passive$$

As we will show with the Lotus Notes case study, the first extension provides for modular data structures, the second extension provides for modular programs.

In general the expressiveness of a language greatly increases with higher-orderness. However, the increased complexity of the language itself, due to adding higher-order features, is very limited. This contrasts with the greatly increased complexity of the systems that can be described with higher-order languages.

Rules and programs revisited

As a consequence of higher-orderness we have to extend the rule syntax. We introduce additional *names* for each of the multisets used in the program. Pattern matching is thereby restricted to the appropriate multisets. Moreover, patterns may contain configurations.

$$\begin{aligned} reaction &\equiv pattern_1{:}name_1, \ldots, pattern_n{:}name_n \\ action &\equiv pattern_1{:}name_1, \ldots, pattern_n{:}name_n \\ pattern &\equiv \_ \mid value \mid Var \mid (pattern_1, \ldots, pattern_n) \mid [Var, pattern] \end{aligned}$$

As another consequence of higher-orderness program composition operators have, strictly speaking, become redundant. Indeed, it is shown in [6] that the sequential and parallel composition operators can be implemented in the higher-order language itself.

4. Specifying the Replication Algorithm

In this section we give informal and formal specifications for each of the three steps of the replication algorithm. As we proceed, we introduce our specification formalism based on higher-order multiset rewriting. In the next section we give a systematic overview of the formalism.

At first we concentrate on the replication between two computers. In the final subsection of this section we extend the specification to include repeated replications with several remote computers.

First step

The designers of Lotus Notes wrote a compact specification of their replication algorithm in structured English [5]. The first step is elaborated as:

> *1. Create a list of databases (requiring replication):*
>
> *Find the databases common to both local and remote computers, and for each database, verify that the remote database has been modified since the last replication with the local computer.*

What do we minimally need to model this step? Clearly the local and remote computers must be represented with their respective databases. Therefore we create two multisets, loc_1 and rem_1. Each element of these multisets represents a database: *id* identifies the database, *docs* represents all the documents contained in the database (*docs* will be further elaborated in the next step), and t is a timestamp indicating the most recent replica.

$$loc_1 \equiv rem_1 \equiv \{(id, docs, t)\}$$

Furthermore we have to create a list for the databases shared between both local and remote computers. This list is also modeled as a multiset, $list_1$.

The first step of the algorithm can be specified with just one rewrite rule on the contents of the multisets. Rule $step_1$ filters all databases which are shared between the local and the remote computer. We take a database (I_l, D_l, T_l) from the local computer loc_1 and a database (I_r, D_r, T_r) from the remote computer rem_1. If both databases have the same identifier ($I_l = I_r$), then we have found a shared database. If additionally the replica of the database on the remote computer has been modified since the most recent modification of the local replica ($T_r > T_l$), then we put all information pertaining to this database in the intermediate multiset $list_1$ for further processing by the next steps. Last but not least, the one-way pull requirement, which states that the remote replica remains unchanged, forces us to restore rem_1 by reinserting (I_r, D_r, T_r).

$$\begin{array}{c} step_1 \equiv \\ (I_l, D_l, T_l){:}loc_1, (I_r, D_r, T_r){:}rem_1 \rightarrow \\ (I_l, D_l, T_l, D_r, T_r){:}list_1, (I_r, D_r, T_r){:}rem_1 \leftarrow \\ I_l = I_r \wedge T_r > T_l \end{array}$$

When rule $step_1$ is no longer applicable, we are sure that $list_1$ contains all shared databases for which the remote replicate is newer. Furthermore we know that loc_1 continues to hold all of the databases which are either not available on the remote computer or for which the remote replicas are outdated. Finally we can be sure that the remote replica has not been modified.

Note: It is common practice to eliminate explicit tests for equality of two variables by substituting a single variable for both (equal) variables. Replacing I_l and I_r by I thus allows us to eliminate the $I_l = I_r$ condition in $step_1$. Note however that such transformations are nothing but syntactic sugar.

$$\begin{array}{c} step_1 \equiv \\ (I, D_l, T_l){:}loc_1, (I, D_r, T_r){:}rem_1 \rightarrow \\ (I, D_l, T_l, D_r, T_r){:}list_1, (I, D_r, T_r){:}rem_1 \leftarrow \\ T_r > T_l \end{array}$$

Second step

In the first step of the replication algorithm we checked for shared databases between the local and remote computers. In the second step we check for shared documents within a shared database. The shared database is identified by *id*. The local and remote replicas are modeled by the loc_2 and rem_2 multisets; *prev* and *pres* specify their respective, most recent replication times.

The structure of documents is analogous to the structure of databases. Both have an identifier and a timestamp indicating the most recent modification. Databases are decomposed into documents. We do not detail the deeper structure of documents. All we need is the ability to copy the content *cont* of any type of document, be it text or images or audio or whatever.

In practice Lotus Notes implementations will decompose the contents of a document recursively and provide specialized copying algorithms for different document types. However, the replication algorithm abstracts away these further levels.

$$docs \equiv loc_2 \equiv rem_2 \equiv list_2 \equiv \{(id, cont, t)\}$$
$$prev \equiv pres \equiv t$$

The informal specification of the second step of the replication algorithm consists of two substeps, one for the remote replica of the database and one for the local replica.

> *2. Create a list of documents (requiring replication):*
>
> *a. Open the remote database and create a list of all the documents that have been modified since the last replication. For each document include its document identifier and its last modification time.*
>
> *b. Open the local database and create a list of all the documents.*

In the first substep we have to filter out those documents of the remote replica of the database that are more recent than *prev*, the last modification time of the local replica. All documents that fulfill this condition are collected into the new multiset $list_2$. Unfortunately those documents are at the same time removed from rem_2. Several solutions are conceivable, most of them requiring an additional intermediate multiset and one or more rules for restoring rem_2 to its original state. We will present a compact solution in section 5.

$$step_{2a} \equiv (I, C_r, T_r){:}rem_2, T{:}prev \rightarrow (I, C_r, T_r){:}list_2, T{:}prev \leftarrow T_r > T$$

For the second substep we do not even need a rewrite rule because the loc_2 multiset already contains all the documents of the local database. In other words $step_{2b}$ is empty. As a result the specification of the entire second step, which consists of the sequential execution of substep $step_{2b}$ after substep $step_{2a}$, is reduced to the first substep only.

$$step_2 \equiv (step_{2b} \circ step_{2a}) \equiv step_{2a}$$

Third step

The informal specification of the third step does the actual copying of the newer document versions from the remote to the local replica. It compares the timestamps of both versions and contains a substep for each of the possible cases.

> *3. Replicate listed documents:*
>
> *For each entry in the local list, find the corresponding entry in the remote list.*

a. If the document in the remote database is a newer version than the version in the local database then copy the document to the local database.

b. If the document in the remote database is marked as deleted then delete the document in the local database.

c. If the document in the remote database is older than the version in the local database or the document is not in the remote database then do nothing since the remote computer will copy the document from the local database.}

d. For the remaining documents in the remote database list, copy them to the local database since these are new documents.

Whether the remote version is newer or older than the local version depends on the comparison of their respective timestamps T_r and T_l. In $step_{3a}$ we overwrite the local version in loc_2, whereas in the complementary case of $step_{3c}$ we maintain the local version.

$$step_{3a} \equiv (I,\_,T_l){:}loc_2,(I,C_r,T_r){:}list_2 \rightarrow (I,C_r,T_r){:}loc_2 \leftarrow T_r > T_l$$
$$step_{3c} \equiv (I,C_l,T_l){:}loc_2,(I,\_,T_r){:}list_2 \rightarrow (I,C_l,T_l){:}loc_2 \leftarrow T_r < T_l$$

Note: The underscores in the element patterns of the rewrite rules indicate *don't cares*. Don't cares substitute variables whose actual values are irrelevant in the remainder of the rule. Again, this is mere syntactic sugar.

In the case corresponding to substep $step_{3b}$, we mark a document as deleted by giving its timestamp the reserved value *del*. Note that the body of $step_{3b}$ is empty.

$$step_{3b} \equiv (I,C_l,T_l){:}loc_2,(I,\_,T_r){:}list_2 \leftarrow T_r = del$$

Finally, the last case is a *catch-all*: it applies to all remaining elements of $list_2$. When $step_{3d}$ is no longer applicable, $list_2$ will be empty. Note that the condition part of the rewrite rule $step_{3d}$ is empty.

$$step_{3d} \equiv (I,C_r,T_r){:}list_2 \rightarrow (I,C_r,T_r){:}loc_2$$

We obtain the complete program for $step_3$ by sequential composition of all four substeps (The brackets, which do not influence execution order, underline the special nature of the last substep).

$$step_3 \equiv \left(step_{3d} \circ \left(step_{3c} \circ step_{3b} \circ step_{3a}\right)\right)$$

Complete algorithm

The hierarchical object structure, with databases on top and documents below, dictates the structure of the replication algorithm. $step_1$ operates at the top level, whereas $step_2$ and $step_3$ are situated below. The concept of configuration helps us in modeling this hierarchy.

Taken together, the three multisets (loc_1, rem_1, and $list_1$) structure of the top level of our specification. The actual *logic*, or control, of the specification is captured by a program $progr_1$.

$$level_1 \equiv [progr_1, loc_1, rem_1, list_1]$$

As we descend one level in the object hierarchy we also descend one level in the specification. The $level_2$ configuration summarizes the multisets and rules relevant for the lower level (A numerical subscript indicates the level of the specification to which a specific
multiset belongs).

$$level_2 \equiv [progr_2, id, loc_2, prev, rem_2, pres, list_2]$$

The link between the two levels is made by hierarchical inclusion of configurations, as illustrated by the adapted rule for $step_1$.

$$\begin{gathered} step_1 \equiv \\ (I, D_l, T_l){:}loc_1, (I, D_r, T_r){:}rem_1 \rightarrow \\ [progr_2, I, D_l; T_l, D_r, T_r, \varnothing]{:}list_1, (I, D_r, T_r){:}rem_1 \leftarrow \\ T_r > T_l \end{gathered}$$

The program for the lower level, $progr_1$, is a straightforward sequential composition of $step_2$ and $step_3$.

$$progr_2 \equiv (step_3 \circ step_2)$$

The program at the top level, $progr_2$, consists of $step_1$ followed by a rule which recovers embedded configurations from the lower level when they turn passive. There is no corresponding task in the informal description, mainly because the informal description is incomplete: It does not explicitly indicate that the second and third step of the algorithm should be repeated *for each database* selected by the first step.

$$progr_1 \equiv (step_4 \circ step_1)$$
$$step_4 \equiv (I, D_l, \_, \_, T_r, \varnothing){:}list_1 \rightarrow (I, D_l, T_r){:}loc_1$$

Note on modification times

In a setting where a local computer replicates with more than one remote computer, replicas of databases (or versions of documents) cannot be uniquely identified by a single timestamp. The Notes system relies on replication tables which contain one timestamp for each remote computer. The effect of this complication on the replication algorithm is minimal: When a timestamp is required, it is selected from a multiset that models the replication table (using an identifier for the remote computer).

In practice versions of Notes documents are identified by a timestamp and an additional version number. The version number is needed to resolve the conflicts that may occur when the clocks of several distributed computers are not sufficiently synchronized. The specification of such complex identifiers remains outside the scope of this report because they have no further effect on the general structure of the replication algorithm.

5. Formal vs. Informal Specifications

In section 4 we went to great lengths to show that the replication algorithm can be specified with rewrite rules. What has been lacking is a clear motivation for why we prefer the formal specification over the informal description in structured English.

In this section we discuss the benefits of being formal: we highlight an error in the informal specification, eliminate redundant data structures, and expose the natural parallelism of the replication algorithm.

Detecting errors

Consider the third step of the algorithm. What happens when the remote and the local replica of the database both hold the same version of a document? Clearly the remote version is not older ($step_{3a}$), nor is it newer ($step_{3c}$), nor deleted ($step_{3b}$). So the catch-all ($step_{3d}$) would apply, meaning that the document is treated as new, and thus copied to the local replica. But the document is not new, it was already present locally. Consequently, after one replication the local replica holds two copies of the same document; after n replications, it holds $n+1$ copies!

When we develop a formal specification for the case structure of the third step, the omission of the "same version" case is immediately apparent. It suffices to amend the condition part of the $step_{3c}$ rule to obtain a correct specification.

$$step_{3c} \equiv (I,C_l,T_l):loc_2,(I,\_,T_r):list_2 \rightarrow (I,C_l,T_l):loc_2 \leftarrow T_r \leq T_l$$

Such, admittedly small, errors are easily overlooked when writing natural language specifications. Terms like "older" and "newer" are too vague. In contrast mathematical formulations such as "greater then" and "greater or equal then" are unambiguous.

Another obscurity in the informal specification, namely the lack of an explicit "for each" loop around the second and third steps, has already been resolved at the end of section 4.

Eliminating redundant lists

At each level of the informal specification lists are introduced for holding intermediate data sets. There is however no indication in the remainder of the specification why a list structure would be more appropriate than any other container type. The intermediate data sets are not ordered, nor is there any other relation between their elements justifying the linearity a list structure imposes. For the purposes of a high-level specification the least restrictive container type, the multiset, is undoubtedly more suitable. We nevertheless continue to call the intermediate multisets lists, in order to illustrate the correspondence with the informal specificâtion.

Given a little experience in analyzing rule-based programs, it becomes quickly apparent that the entire intermediate multisets, not just their list structures, are redundant. The elements we put in $list_1$ in rule $step_1$ might as well be put directly in loc_1.

$$\begin{aligned} step_1 &\equiv (I,D_l,T_l){:}loc_1,(I,D_r,T_r){:}rem_1 \to \\ &\quad [(step_3+step_2),I,D_l,T_l,D_r,T_r]{:}loc_1,(I,D_r,T_r){:}rem_1 \leftarrow T_r > T_l \\ step_4 &\equiv (I,D_l,\_,\_,T_r,\varnothing){:}loc_1 \to (I,D_l,T_r){:}loc_1 \end{aligned}$$

Additionally we can eliminate $list_2$. Instead of collecting the filtered elements of rem_2 in $list_2$, we annotate those elements of rem_2 with the reserved value *filt*. At the same time this annotation helps us in recovering the original content of rem_2. Rules $step_{3b}$ and $step_{3c}$ undergo similar adaptations.

$$\begin{aligned} step_{3a} &\equiv (I,\_,T_l){:}loc_2,(filt,I,C_r,T_r){:}rem_2 \to \\ &\quad (I,C_r,T_r){:}loc_2,(I,C_r,T_r){:}rem_2 \leftarrow T_r > T_l \\ step_{3d} &\equiv (filt,I,C_r,T_r){:}rem_2 \to (I,C_r,T_r){:}loc_2,(filt,I,C_r,T_r){:}rem_2 \end{aligned}$$

Exposing natural parallelism

The lists are just one instance of the sequential bias of the original, informal specification of the replication algorithm. The overall structure of the informal specification, with sequentially numbered steps, is another instance of that bias.

It is not our intention to blame the authors of the original specification for its artificial sequentiality, because their target architecture really was a distributed set of sequential computers. Nevertheless a specification style which does not impose unnecessary sequentiality is preferable because it opens up the algorithm to a wider range of applications, especially those that are not apparent from the outset.

Most of the sequentiality imposed by the informal specification can be relaxed. At the top level there is no reason why $progr_2$, which copies documents between two replicas of the same database, could not run in parallel with $progr_1$, which looks for databases with diverging document versions. In fact the use of embedded configurations already captures the parallelism between $progr_1$ and $progr_2$. As it turns out, it would be far more difficult to specify a sequential behavior. At the risk of repeating ourselves, this is a perfect illustration of the parallel bias characterizing multiset rewrite specifications.

In addition, at the lower level, within $progr_2$, we can pipeline $step_2$ and $step_3$. Consequently documents belonging to different databases can be copied in parallel.

$$progr_2 \equiv (step_3 + step_2) \geq (step_3 \circ step_2)$$

A final refinement allows for parallelism between the substeps of $step_3$. The execution order of the first three cases is clearly irrelevant. Not so for the catch-all however: $step_{3d}$ must be executed after the others. So a little sequentiality is unavoidable.

$$step_3 \equiv \left(step_{3d} \circ \left(step_{3c} + step_{3b} + step_{3a}\right)\right) \geq \left(step_{3d} \circ \left(step_{3c} \circ step_{3b} \circ step_{3a}\right)\right)$$

6. Discussion

Concretely, the example of replication in Lotus Notes has shown how multiset rewrite specifications expose the inherent parallelism of distributed systems. Parallel implementations, conforming to our formal specifications, might conceivably serve as the basis for highly available object servers. We anticipate a golden future for similar algorithms on the Internet where the business case for replicated servers is getting stronger by the day, not in the least due to the explosive use of facilities like the World-Wide Web. In fact, Irene Greif of Lotus Development already suggested as much in her invited talk at CSCW'94.

Moreover we have illustrated how the higher-order features of multiset rewrite specifications model hierarchical object composition and promote modularization. Already for the relatively simple replication algorithm we reap these benefits: without modularization many of the transformations presented in this report would have remained undetected. Imagine what multiset rewriting specifications might bring for more complex cases, such as transaction-based systems.

What is missing from this report is a comparison of higher-order multiset rewriting with other specification techniques. Whereas multiset rewriting is particularly successful at capturing the structural aspects and the inherent parallelism of distributed systems, it is still unclear whether such specifications can be used for proving properties of such systems. This is definitely a topic deserving further research.

Acknowledgements

Section 2 liberally borrows parts of the description of Lotus Notes from [5]. Section 3 relies on results from [6].

Detailed comments made by Chris Hankin and David Sands where very helpful for our research on multiset rewrite systems as specification tools. Many thanks also go to Kees Schuerman and Bent Thomsen for proof reading an earlier version of this report.

The work reported here is partly sponsored by Esprit Basic Research Action 9102: Coordination.

Bibliography

[1] J-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, (1), 1993.

[2] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 1992.

[3] M. Bourgois. Specifying a distributed and reflective implementation of LO in higher-order gamma. In *Proceedings of the Geneva Coordination Workshop.* To be published. IC Press, 1995.

[4] C. Hankin, D. Le Métayer, and D. Sands. A calculus of gamma programs. In *Languages and Compilers for Parallel Computing, 5th International Workshop* (LNCS 757), Springer Verlag, 1992.

[5] L. Kawell, S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. In *CSCW'88*, Portland, Oregon.

[6] D. Le Métayer. Higher-order multiset programming. In *DIMACS workshop on specification of parallel algorithms.* American Mathematical Society, 1994.