Specification of Convergent Abstract Data Types for Autonomous Mobile Computing

Carlos Baquero Francisco Moura

{cbm,fsm}@.di.uminho.pt
http://gama.di.uminho.pt/

Universidade do Minho
Departamento de Informática
Largo do Paço
4700 Braga
Portugal

2nd October 1997

Abstract

Traditional replica control mechanisms such as quorum consensus, primary replicas and other strong consistency approaches are unable to provide a useful level of availability on unconstrained mobile environments. We define an environment thats exploits pair-wise communication and allows autonomous creation and joining of replicas while ensuring eventual convergence. A set of composable components (ADTs) are formally specified using the SETS Calculus. These components can be used to build simple distributed applications that take advantage of peer-to-peer communication between mobile hosts.

Keywords: Mobile Computing, ADTs, Replication, SETS Calculus

1 Introduction

In recent years distributed systems have evolved considerably. Two examples are mobile computing [11], prompted by the increasing presence of portable an handheld computers, and large-scale distribution on the internet [6, 5]. In both cases programmers have now to deal explicitly with the absence of communication, that is, disconnection or network partitions should not be regarded as occasional faults but should be considered early in the design phases as they can occur frequently or during long periods. This in contrast with traditional client-server systems where communication among machines can be expected most of the time.

Furthermore, in the few cases where lack of communication is taken into consideration and replication is used to increase availability, the legacy of the client-server model is still there. For example, notebook users will try to connect to a remote, centralized server (e.g. Lotus Notes), and the techniques for achieving data consistency often require a home site.

In this paper we claim that this "all-or-nothing" approach to replica control does not exploit the full potential of distributed systems and mobile computing in particular. We are prepared to sacrifice consistency for availability and allow replicas to fork new copies or to merge with other replicas in a rather loose or *autonomous* way. Replication and merging only require peer-to-peer communication. Our vision is that a replicated object will eventually be consistent when all replicas finally merge. Strong consistency is replaced by pair-wise convergence.

This work is in line with recent projects [3, 12, 10, 13, 9] that try to support a free pattern of mobility where information can be exchanged whenever two hosts meet and communication with a centralized network or server is only a special case of peer-to-peer interaction. As a consequence, the focus now shifts from strong consistency among a group of known replicas towards pair-wise consistency of information scattered through an arbitrary number of replicas of an abstract entity.

Clearly, since applications are allowed to alter the state of the local replicas, one must be sure that all replicas can in fact be merged without aborting operations or loosing any of the updates made to the replicas. This is where specification comes in. In this paper we present some abstract data types that guarantee pair-wise merging of the data. These data types are composable, and can be used to build simple distributed applications that explore the casual or deliberate encounter of mobile hosts.

The acceptable data types are strictly driven from the minimal requirements for permanent and unconstrained autonomous operation. Although the resulting valid data types are not suitable for aplications with strong consistency needs, they introduce vast possibilities of information sharing and re-integration for areas such as: Personal information management applications on small pocket devices that have traditionally a policy of no-sharing and permanent availability, for example, phonebooks, organizers; Convergence of legacy distributed data (generated by user mobility) that can conform to a suitable merge semantics, for example, bookmarks, mailboxes, newsrc; New mobile applications that aim at permanent availability and tailor their shared data types under this minimal model.

2 Environment

We will model the autonomous incorporation of information on replicas of a given abstract entity. Each replica should, at any time, be able to derive new replicas and each two replicas should be capable of converging into a new replica.

2.1 Description

Consider one replicated abstract entity E to which we associate a possibly infinite set of replicas $R(E)$. At any given time we will have k replicas that belong to the set of all potential replicas of E, $\{e_1, \cdots, e_k\} \subset R(E)$. The abstract entity E represents the concept that is replicated, for instance "The 5 banks with the best interest for house loans" or "The set of known prime numbers".

Each replica is bound to a abstract entity, has a unique identifier, conforms to a given *abstract data type* (shortly *type*) and stores data that conforms to a state description in the type, $Replica \equiv Entity \times Type \times Id \times Storage$. A type defines the replica state and signature, and specifies operations in terms of manipulations of the state, input elements and output elements, $Type \equiv Name \times Sig \times StateDef \times OpsDef^*$.

There is a one-to-many (or at least one-to-one) relationship from *Entity* to *Replica* and from *Type* to *Replica*. Replicas of the same type can model different abstract entities. For instance, the two entities "The set of known prime numbers" and "The set of known composite numbers" could be modeled by the same ADT, one that stores growing sets of positive integers.

A initial replica can be created by a special operation *Init* that constructs a replica from a abstract entity and a type (we will assume, conceptually, that all replicas of a given entity will derive from a single initial replica, although, in practice, for some data types we can autonomously create new replicas. c.f. IncSet)

Init: *A special operation Init defined on each type creates a initial replica e_i from a type T and a abstract entity E. This is denoted by* $\xrightarrow{Init(T,E)} e_1$.

This operation adds e_i to the pool of existent replicas. Physically the pool of replicas can be scattered through different locations and each replica is represented by a finite data sequence (for instance, like marshaled object instances).

Replicas do not have processing capability. In order to change the information stored on a replica a process must be associated to it and apply an operation on the replica. We do not force replicas to be associated to specific mobile nodes, so any mobile node can actuate on any replica and, consequently, the replica identity must be independent of the mobile node identity. When an operation is applied to a replica the replica might change, by changing its *Storage*, to accomodate the new information. This derives a non-strict partial order, denoted by $\preceq$, which expresses **incorporation** of information[1], mapping the evolution of the state stored at the replicas.

Each type defines the $\preceq$ relation by a boolean valued operation *Leq* that relates two replicas. This operation allows the validation of the ADT operations on respect to the partial order requirements that will be expressed.

We can now define incorporation.

Incorporation: *Given a replica e_x, by issuing an operation op on this replica and*

[1] This evolution of the replica state by incorporation of information also occurs when joining replicas with diferent states, as will be explained latter when introducing the join operation.

now referring the replica as e'_x to indicate the potential change on the storage, we have the transformation $e_x \xrightarrow{op(e_x)} e'_x$, and say that $e_x \preceq e'_x$.

The *Storage* is the only replica section allowed to change by incorporations, and these changes might not affect the outcome of the *Leq* evaluation. Some operations (read-only operations) might not even change the *Storage*. Thus incorporation is not required, or expected, to derive a strict partial order $\prec$.

The binary relation $\preceq$ on replicas, being a non-strict partial order, holds the normal **reflexivity**, **antisymmetry** and **transitivity** conditions. Equality of replicas, denoted by the binary relation $\simeq$, is driven from the antisymmetry condition, since $e_x \preceq e_y \wedge e_y \preceq e_x$ implies $e_x \simeq e_y$.

Each replica can produce new replicas by the special operation *Fork* denoted by the symbol $\times$ and defined on the replica type. The replica that receives this operation ceases to exist and two replicas are generated.

Fork: *Issuing a special operation $\times$ on a replica e_j, this replica is replaced by two new replicas e_k and e_l, which is denoted by $e_j \xrightarrow{\times(e_j)} e_k, e_l$, so that $e_j \simeq e_k \simeq e_l$.*

Forking preserves the *Storage* and consequently the outcome of the *Leq* operation. We denote this by relating the equivalent states by $\simeq$.

It is now appropriate to say that not all replicas need a unique *Id*, the behavior of some types allows *Init* special operations that generate a undefined id $\perp$. These undefined ids are kept upon forks. On the other hand, those types that require unique ids must derive the adequate ids on their definition of the special *Fork* operation.

The convergence of replicas is achieved by the operation *Join,* denoted by the symbol $+$, and also defined on the type associated to the replica. When two replicas join, they cease to exist and produce a new replica with a distinct identifier, or with undefined identifiers (when applicable). The join of two replicas potentially creates a new *Id* and generates a new *Storage* by applying a suitable merge to the two replicas *Storage*.

Join: *Issuing a special operation $+$ on any two replicas e_x and e_y, we obtain a new replica e_z that is the simplest replica that incorporates the two replicas. This is denoted by $e_x, e_y \xrightarrow{+(e_x,e_y)} e_z$, so that: $e_x \preceq e_z \wedge e_y \preceq e_z$; and $\forall e_i : e_x \preceq e_i \wedge e_y \preceq e_i \Rightarrow e_z \preceq e_i$.*

This definition of *Join* that computes a merge for all replica pairs is in fact the definition of a **join** on a partially ordered set (traditionally written as $x \vee y$ for the join of the elements x, y). Since we are able to derive the **join** of any two replicas, it defines a **semi-lattice**. The existence of a lattice would require a **meet** for any two replicas, which is not relevant for this concrete framework.

The axiomatization here introduced for the special operations {*Init*,*Fork*,*Join*} and for normal operations on *Incorporation*, is strictly driven from the assumptions on the underlying mobility pattern. The aim is to establish grounds for autonomous creation, use, replication and convergence of replicas, under a minimal set of requisites.

These assumptions would also call for some extra properties that ensure that the two special operations {*Fork*,*Join*} do not generate by themselves any information and that they are immune to some order considerations. The interpretation of these additional properties provides a sound modeling of the underlying environment, and remove indeterminism.

Since our definition of *Join* produces a join semi-lattice, these needed properties are in fact normal laws of semi-lattices and do not need further proof.

Idempotence: $+(x, x) \simeq x$, also written as $x \vee x \simeq x$
Commutativity: $+(x, y) \simeq +(y, x)$, also written as $x \vee y \simeq y \vee x$
Associativity: $+(x, +(y, z)) \simeq +(+(x, y), z)$, also written as $x \vee (y \vee z) \simeq (x \vee y) \vee z$

This Idempotence condition shows that the special operation *Join* does not introduce spurious information when computing the merge of the two storages. Arbitrary sequences of forks and joins with no interliving incorporations do not change the replica on respect to the partial order. Commutativity and Associativity verification ensures that the order in which joins are performed does not interfere with the final resulting merge when there are no interliving incorporations.

Each ADT, once specified, can be verified for compliance with these properties. This article will not deal with the verification process but a proof procedure for a concrete component is sketched in appendix.

2.2 Additional assumptions

Together with a naming scheme that can autonomously assign new identifiers to the replicas generated by the special operations, the assumed properties allow a replication environment in which replicas can be stored on any persistent medium and supporting arbitrarily large numbers of replicas than can be autonomously generated by iterated forking from any replica.

In this framework we express that all collected information is relevant and that no two replicas can collect conflicting information. We assume no other ordering to the operations than registered causality (registered along forks, joins and incorporations). There is also no concept of global time, although specific reconciliators could use time sources to merge concurrent traces of operations[2].

2.3 Trading strong consistency for permanent conflict free availability

Lack of strong consistency might appear as the major drawback of this environment, but under unconstrained mobility it is not possible to have both strong consistency and permanent availability, and availability is the reason to carry a mobile device. There are no bounds to the time that a mobile node can be outside a given network and information

[2]Naturally, this potential external time source (for instance, by having GPS time source on all processing nodes) is not totally reliable and should not conflict with the registered causality.

exchange is likely to be limited to occasional peer-to-peer interactions. Although these factors apparently restrict the set of applications that can operate in this environment, the environment itself mirrors human centered interactions where users seldom have the need for total consistency among the information they hold [12].

Applications for personal information management (PIM) have for long be used in small handheld devices and rely almost completely on unshared information. Any amount of information sharing (replication of a common abstract entity) that does not hamper mobility and constant availability can greatly improve the scope of traditional PIM applications.

The components and structuring properties presented in this paper allow the description of complex convergent types. Any target application that can fit a potentially sharable part of its data under this description is granted with the appropriate sharing and merging mechanisms.

3 Classification

We will now present a set of components that respect the convergence properties introduced in the previous section. The specification notation is adapted from the SETS Calculus [7, 8] [3] and models components by their state and by the operations that interact with the state. The notation used under SETS is rooted on standard set-theory which should enable an intuitive reading of the descriptions.

Component types are described under four sections, **Type**, **Interface**, **State** and **Model**, which map the elements that define *Type*, respectively, *Name*, *Sig*, *StateDef* and $OpsDef^*$. These sections have the necessary information for generating *Replica* instances with *Entity*, *Type*, *Id* and *Storage*.

The **Type** section introduces the name of the component, thus declaring a new type name. Reuse of specifications along the hierarchy is done by adding "$<$ *SuperType*" to the type name, which indicates reuse of operation definitions from the ascending type chain. Inherited operation definitions can be redefined on the subtype.

Replicas store the *Type* name and the *Entity* name. These labels will not be treated as first class entities on the component specifications to avoid cluttering the description, since they are properties that are created with the initial replica and are fixed along the replica's life. However this labels can be accessed with two functions, with corresponding names, $type(\ldots)$ and $entity(\ldots)$. Apart from some potential casting along types, that will not be addressed, it is assumed that only replicas of the same *Type* and reflecting the same *Entity* are subject to joins and comparable. This is captured in a *match* function, $match(c_1, c_2, b') : b' = (type(c_1) = type(c_2) \wedge entity(c_1) = entity(c_2))$. In this notation b' is a output symbol that is computed from the inputs c_1 and c_2.

The **Interface** section groups the signature of the *normal operations* that the type holds, including the inherited ones (which are represented for better reading). These signatures show the input and output arguments, but do not refer the component state.

[3] The SETS home page is at http://www.di.uminho.pt/~jno/html/setshp.html

State indicates a model for the *Storage* and *Id*. The *Storage* model is associated to the symbol Σ, and the *Id* model to I.

Operations and special operations are grouped under the section **Model** and might introduce input and output symbols, with output symbols decorated with a apostrofe. The value of the *Storage* (Σ) before and after the operation are respectively denoted by σ and σ', and similarly ι and ι' are used for the *Id* (I) values.

Components are grouped on a hierarchy driven by the specification of their state (*Storage* model).

4 Components

The components hierarchy is topped with the **Basic** component. This component hosts a unstructured state (storage), and provides some basic functionalities common to all the components. The supported operations ensure that the component is only replicated after the state has been initialized. This component has no inspective operations defined. Thus the first extension of this component, depicted in the **Const** component, adds a read operation over the state.

Other extensions would permit changes to the state as long as the component has not been replicated. Only under these conditions it is sound to keep the basic *Join* special operation as defined in **Basic**.

4.1 Basic and Const components

Type: Basic
Interface:
$Write : Storage$
State:
$\Sigma = X, I = Y$
Model:
$Init(\sigma', \iota')$
post $\sigma' = \bot, \iota' = \bot$

$Write(\sigma, a, \sigma')$
pre $\sigma = \bot$
post $\sigma' = a$

$Fork(\sigma, \iota, \sigma'_l, \iota'_l, \sigma'_r, \iota'_r)$
pre $\sigma \neq \bot$
post $\sigma'_l = \sigma'_r = \sigma, \iota'_l = \iota'_r = \iota$

$Join(\sigma_l, \iota_l, \sigma_r, \iota_r, \sigma', \iota')$
pre $\sigma_l = \sigma_r$
pos $\sigma' = \sigma_l = \sigma_r$
$\iota' = \iota_l = \iota_r$

One singular instance of **Basic** is symbolized by τ and acts as a fixpoint for recursive definitions of *Join*, $\tau, \tau \xrightarrow{+(\tau,\tau)} \tau$.

When the symbol C is used on a *Storage* model it symbolizes a polymorphic reference to **Basic**. This reference is matched by any *component,* which means that it symbolizes a generic placeholder for arbitrary instances, including τ. C is used for type dependent recursive invocations of the special operations *Join* and *Fork*, and is seen in the **ConstSeq** and **IncMap** components.

Type: Const $<$ Basic
Interface
$Write : Storage$
$Read :\rightarrow Storage$
State:
$\Sigma = X, I = Y$
Model:
$Read(\sigma, a')$
pre $\sigma \neq \perp$
post $a' = \sigma$

$Leq(\sigma_l, \sigma_r, b')$
post $\begin{cases} \sigma_l = \sigma_r \Rightarrow b' = true \\ else \Rightarrow b' = false \end{cases}$

Const replicas model the dissemination of constant information. This component is useful when combined with other convergent components as it can model, for instance, constant headers on mutable files (such as the 8 lines header and 1 line footer on the Netscape bookmarks file).

The **Const** component inherits from **Basic** its four (3 special +1 normal) operations and adds a *Read* operation. Notice that **Basic** does not assign unique identifiers (stored under I) upon the special operations. In fact, most components do not need to distinguish replicas, and not all components need the same level of labeling.

The ability to distinguish replicas is sometimes needed to identify their common past, in order to avoid duplication of incorporated data (for instance due to non-idempotent write operations). However, most of the ADTs presented here are immune to this operation duplication. By looking at the dependencies among the operations, we are currently classifying the factors that bound the need to distinguish replicas.

4.2 Sets

This family groups components that are based on the functor 2^X i.e. $X \rightarrow \{0, 1\}$, for finite instances of X. As such this family works with subsets of X. Operations on these subsets must be constrained in order to ensure convergence. From the possible components fitting under this family we will concentrate on two particular cases, respectively a set that only grows and a set that can model distributed mailbox handling.

These two sets redefine *Init* so that empty sets are the initial storage, this has the consequence of inhibiting *Write* as its pre-condition is $\sigma = \perp$. Nevertheless for shrink only sets, which are not shown here, *Write* plays a central role on the set initialization.

Type: IncSet $<$ Basic
Interface:
$Insert : Elem$
$Find : Elem \rightarrow Bool$
State:
$\Sigma = 2^X, I = Y$
Model:
$Init(\sigma', \iota')$
post $\sigma' = \oslash, \iota' = \perp$

$Insert(\sigma, e, \sigma')$
post $\sigma' = \sigma \cup \{e\}$

$Find(\sigma, e, b')$
post $\begin{cases} e \in \sigma \Rightarrow b' = true \\ e \notin \sigma \Rightarrow b' = false \end{cases}$

$Join(\sigma_l, \iota_l, \sigma_r, \iota_r, \sigma', \iota')$
post $\sigma' = \sigma_l \cup \sigma_r$
$\iota' = \iota_l = \iota_r$

$Leq(\sigma_l, \sigma_r, b')$
post $\begin{cases} \sigma_l \subseteq \sigma_r \Rightarrow b' = true \\ else \Rightarrow b' = false \end{cases}$

Growing sets model situations where data is autonomously and incrementally gathered. As expected all collected data is relevant, and upon a join the resulting replica holds all elements. This component, in particular, starts with a empty set (defined upon *Init*) but we could have a growing set that starts with any initial state supplied upon a singular write operation[4].

Most of these structures can grow indefinitely and thus lack some kind of growth control policy. This depends on the application and can use techniques such as assigning *time to live* stamps to the whole set or to the set elements, or by allowing the creation, at some time, of a anihilater replica that will act as the absorbent element of the join.

Conceptually the next component, **IncDecSet,** is a single set when viewed through its inspection operations, but its state integrates two sets.

Type: IncDecSet $<$ Basic
Interface:
$Insert : Elem$
$Erase : Elem$
$Find : Elem \rightarrow Bool$

[4]For this we would need some extra preconditions on the operations *Insert* and *Find*, and some other minor changes.

State:
$\Sigma = 2^X \times 2^X, I = Y$
Model:
$Init(\sigma'_{in} \times \sigma'_{del}, \iota')$
post $\sigma'_{in} \times \sigma'_{del} = \oslash \times \oslash, \iota' = \bot$

$Insert(\sigma_{in} \times \sigma_{del}, e, \sigma'_{in} \times \sigma_{del})$
post $\sigma'_{in} \times \sigma_{del} = (\sigma_{in} \cup \{e\}) \times \sigma_{del}$

$Erase(\sigma_{in} \times \sigma_{del}, e, \sigma'_{in} \times \sigma'_{del})$
pre $e \in \sigma_{in}$
post $\sigma'_{in} \times \sigma'_{del} = (\sigma_{in} \setminus \{e\}) \times (\sigma_{del} \cup \{e\})$

$Find(\sigma_{in} \times \sigma_{del}, e, b')$
post $\begin{cases} e \in \sigma_{in} \Rightarrow b' = true \\ e \notin \sigma_{in} \Rightarrow b' = false \end{cases}$

$Join(\sigma_{l_{in}} \times \sigma_{l_{del}}, \iota_l, \sigma_{r_{in}} \times \sigma_{r_{del}}, \iota_r, \sigma'_{in} \times \sigma'_{del}, \iota')$
post $\sigma'_{in} \times \sigma'_{del} = ((\sigma_{l_{in}} \setminus \sigma_{r_{del}}) \cup (\sigma_{r_{in}} \setminus \sigma_{l_{del}})) \times (\sigma_{l_{del}} \cup \sigma_{r_{del}})$
$\iota' = \iota_l = \iota_r$

$Leq(\sigma_{l_{in}} \times \sigma_{l_{del}}, \sigma_{r_{in}} \times \sigma_{r_{del}}, b')$
post $\begin{cases} \sigma_{l_{out}} \subseteq \sigma_{r_{out}} \wedge \sigma_{l_{in}} \subseteq \sigma_{r_{in}} \Rightarrow b' = true \\ else \Rightarrow b' = false \end{cases}$

IncDecSets keep track of removals and ensures that we can only remove elements that were in the inclusion set at the time of removal. This makes removal dependent from insertion. When a replica that holds a given element meets one that knows of its removal (which we know that is subsequent to the insertion) we can safely remove the element.

This data type can easily model a arbitrary group of mailboxes that potentially receive the same set of messages (although allowing variable delays). The owner of the replicated mailbox can read and delete any messages, knowing that the reconciliation procedure ensures that the deletions are propagated and that any two mailboxes can synchronize to share missing insertions and deletions.

This component could also be extended to model movement of messages from the generic placeholder into folders, as long as this movement is one way only.

4.3 The Sequence Component

This component is based on the functor C^* and models sequences of C elements. We will deal here only with constant well defined sequences of components[5]. Sequences are used a general abstraction to n-ary cross products.

$\underbrace{C_1 \times \cdots \times C_n} \subset C^*$

The structuring mechanism provided by sequences of heterogeneous components enables the creation of compound structures from available components. These structures,

[5]Using a assymetric properties on replicas, such as primary replicas, we could mix n fixed sequences with one growing sequence.

as expected, ensure well defined special operations by managing their delegation (with potential recursion) to the resident components. By expressing generalized cross products this component controls the derivation of semi-lattices from products of semi-lattices, which is a known property.

Each sequence of components must be formed prior to replication. Joining of sequences is decomposed in the ordered pair-wise join of its components. The following specification of a constant sequence is very simple, as it inherits from **Const** the blind initialization of its state with *Write*. This means that the **ConstSeq** can only be replicated by *Fork* after being initialized with a specific sequence of components, that will remain fixed thereof. A more flexible, alternative, specification would allow the sequence to grow, by a *Cons* operation that appends new components, as long as replication has not been initiated[6].

Type: ConstSeq $<$ Const
Interface:
$Write : Storage$
$Read :\rightarrow Storage$
State:
$\Sigma = C^*, I = Y$ where $C = \Sigma \times I$
Model:
$Init(\sigma', \iota')$
post $\sigma' = \perp, \iota' = \perp$

$Fork(\sigma, \iota, \sigma'_l, \iota'_l, \sigma'_r, \iota'_r)$
pre $\sigma \neq \perp$
post with $1 \leq j \leq length(\sigma)$ let $\sigma(j) = s(j) \times i(j)$

$$\text{in } \sigma'_l = \left(\begin{matrix} j \\ s'_l \times i'_l : Fork(s(j), i(j), s'_l, i'_l, s'_r, i'_r) \end{matrix} \right)$$

$$\sigma'_r = \left(\begin{matrix} j \\ s'_r \times i'_r : Fork(s(j), i(j), s'_l, i'_l, s'_r, i'_r) \end{matrix} \right)$$

$\iota'_l = \iota'_r = \iota$

$Join(\sigma_l, \iota_l, \sigma_r, \iota_r, \sigma', \iota')$
post with $1 \leq j \leq length(\sigma_l) = length(\sigma_r)$ let $\sigma_l(j) = s_l(j) \times i_l(j), \sigma_r(j) = s_r(j) \times i_r(j)$

$$\text{in } \sigma' = \left(\begin{matrix} j \\ s' \times i' : Join(s_l(j), i_l(j), s_r(j), i_r(j), s', i') \end{matrix} \right)$$

$\iota' = \iota_l = \iota_r$

$Leq(\sigma_l, \sigma_r, b')$
post with $1 \leq j \leq length(\sigma_l) = length(\sigma_r)$ let $\sigma_l(j) = s_l(j) \times i_l(j), \sigma_r(j) = s_r(j) \times i_r(j)$

$$\text{in } b' = \bigwedge_j \left(\begin{matrix} j \\ \beta' : Leq(s_l(j), s_r(j), \beta') \end{matrix} \right)$$

Sequences are represented as applications of a initial segment of $I\!N$ into T. Although the sequence *Id* is undefined, its *Storage* stores whole components with the corresponding *Storage* and *Id*. This was done explicitly in order to keep the same syntax (with six

[6]The reference implementation, in Java, for this component defines a initialization phase, prior to replication, in wich the components are inserted in the appropriate order.

parameters) when defining and using the *Fork* and *Join* special operations.

4.4 Maps

From this family, of partial functions, we present a **IncMap** that will provide the second structuring component. This component defines a partial function from components (keys) to components, and normally should be used with **Const** components in its domain. Joining of maps requires knowledge on the type of the actual components of each tuple image, as only pairs of tuples with equal keys and matching elements on their image will perform a join of the image elements to derive a new single tuple. This is supported by using both the key and image type in the relation domain.

Type: IncMap $<$ Basic
Interface:
$Insert : Key \times Component$
$Find : Key \times Type \rightarrow Component$
State:
$\Sigma = X \times T \hookrightarrow C, I = Y$ where $C = \Sigma \times I$
Model:
$Init(\sigma', \iota')$
post $\sigma' = \oslash, \iota' = \bot$

$Insert(\sigma, k, c, \sigma')$
pre $k \times type(c) \notin dom(\sigma)$
post $\sigma' = \sigma \cup \begin{pmatrix} k \times type(c) \\ c \end{pmatrix}$

$Find(\sigma, k \times t, c')$
pre $k \times t \in dom(\sigma)$
post $c' = \sigma(k \times t)$

$Fork(\sigma, \iota, \sigma'_l, \iota'_l, \sigma'_r, \iota'_r)$
pre $\sigma \neq \bot$
post with $a \in dom(\sigma)$ let $\sigma(a) = s(a) \times i(a)$
in $\sigma'_l = \begin{pmatrix} a \\ s'_l \times i'_l : Fork(s(a), i(a), s'_l, i'_l, s'_r, i'_r) \end{pmatrix}$
$\sigma'_r = \begin{pmatrix} a \\ s'_r \times i'_r : Fork(s(a), i(a), s'_l, i'_l, s'_r, i'_r) \end{pmatrix}$
$\iota'_l = \iota'_r = \iota$

$Join(\sigma_l, \iota_l, \sigma_r, \iota_r, \sigma', \iota')$
post with $a \in (dom(\sigma_l) \cap dom(\sigma_r))$ let $\sigma_l(a) = s_l(a) \times i_l(a), \sigma_r(a) = s_r(a) \times i_r(a)$
in $\sigma' = \sigma_l \setminus dom(\sigma_r) \cup \sigma_r \setminus dom(\sigma_l) \cup$
$\bigcup_a \begin{pmatrix} a \\ s' \times i' : Join(s_l(a), i_l(a), s_r(a), i_r(a), s', i') \end{pmatrix}$
$\iota' = \iota_l = i_r$

$Leq(\sigma_l, \sigma_r, b')$

post $b' = dom(\sigma_l) \subseteq dom(\sigma_r) \wedge \bigwedge_a \left(\begin{smallmatrix} a \\ \beta' : Leq(\sigma_l(a), \sigma_r(a), \beta') \end{smallmatrix} \right)_{a \in (dom(\sigma_l) \cap dom(\sigma_r))}$

Maps allow the definition of complex folded structures that can be used both to model file systems and classification trees like the one that structures Netscape bookmarks. This later case depicts one situation in which a strictly additive semantics is perfectly reasonable.

These reconciliation procedures can be used, both in applications that control their replicated state from scratch in order to allow autonomous convergence, or in corrective applications that converge replicas that were generated without support for future convergence. Such cases can be seen on the common occurrence of replication of mailboxes and browser bookmark files on mobile nodes and even among distant fixed nodes.

4.5 The Counter

We conclude the presentation of this sample of convergent components by showing a replicated counter. Each replica of the counter is expected to autonomously collect increments. Due to the lack of Idempotence among the *Inc* operations used on the counter, this component must keep a partial track of its past segments of incorporations in order to avoid duplication of increments upon rejoins.

The nature of the information that the **Counter** accumulates is different from the one that the **IncSet** stores. The latter contains *universal information* from which a single item can be observed by many replicas, while the former has *localized information* for which every observation is unique. This distinction can be derived from the existence or not of idempotence among two subsequent incorporations of the same operation.

Type: IncNat
Interface:
$Inc:$
$Count :\rightarrow I\!N$
State:
$\Sigma = 2^* \hookrightarrow I\!N, I = 2^*$
Model:
$Init(\sigma', \iota')$
post $\sigma' = (\begin{smallmatrix} \langle\rangle \\ 0 \end{smallmatrix}), \iota' = \langle\rangle$

$Inc(\sigma, \sigma')$
post $\sigma' = \sigma \dagger (\begin{smallmatrix} \iota \\ \sigma(\iota)+1 \end{smallmatrix})$

$Fork(\sigma, \iota, \sigma'_l, \iota'_l, \sigma'_r, \iota'_r)$
post $\iota'_l = cons(\iota, 0), \sigma'_l = \sigma \cup (\begin{smallmatrix} cons(\iota,0) \\ 0 \end{smallmatrix})$
$\iota'_r = cons(\iota, 1), \sigma'_r = \sigma \cup (\begin{smallmatrix} cons(\iota,1) \\ 0 \end{smallmatrix})$

$Count(\sigma, n')$
post $n' = \sum_{j \in dom(\sigma)} \sigma(j)$
$Join(\sigma_l, \iota_l, \sigma_r, \iota_r, \sigma', \iota')$
post $\sigma' = \sigma_l \cup \sigma_r, \iota' = \iota_l$

$Leq(\sigma_l, \sigma_r, b')$
post $\begin{cases} \sigma_l \subseteq \sigma_r \Rightarrow b' = true \\ else \Rightarrow b' = false \end{cases}$

The relevant property of this component is that it redefines the *Fork* operation in order to generate a unique identifier, of the form 2^* each time it is replicated. The component stores the active identifier and a map of identifiers to counters. Only the counter associated to the active identifier is used[7]. Unlike some of the previous ones this component must take part on the replication itself. This does not restrict replication but forces it to be done at a high level i.e. blind duplication of the marshaled representation of the state is no longer possible.

5 Related Work

Reconciliation of files according to their semantics was described in [9]. This work described the reconciliation procedures that could automatically reconcile concurrent files in the Ficus[4] peer-to-peer replicated file system. Our component description approach can be used to model the specific reconciliators that where designed for the Ficus project. Rumor[10] inherits Ficus legacy and proceeds in the trend to allow sharing of replicated file systems under a model of unconstrained mobility similar to ours. Apparently Rumor uses version vectors to keep track of conflicts between replicas, and in practice imposes a limit of twenty replicas to handle the size of control data. Tracking all the dependencies among operations done under un-hosted replication is no longer possible and requires a labeling scheme that generalizes the notion of version vector to an unbound number of autonomously generated replicas.

Bayou[13] presents a storage infrastructure for mobile applications where exchanges of information is done with pair-wise communications. Operations such as writes are ordered under a *tentative* order that can evolve into a *committed* order that is achieved by having a primary replica at some specific server. Merging is done on a per-operation basis. When new operations are received by pair-wise exchanges, they are checked for conflicts before being applied. If a conflict arises then a specific merge procedure is executed for applying an alternative incorporation of the conflicting operation. These merge procedures are asymetric as they merge one operation into a tentatively ordered log of operations. This system is somewhat more constrained, as it uses essentially hosted replication on servers (which means that replicas are bound to those servers) but this is compensated by allowing binding of clients to different servers.

The GIM replication system[12] made some steps on assessing that a communication system based on autonomous merging and diffusion of replicas could be used to implement a shared appointment manager. This system used epidemic propagation of replicas (which where the actual messages) between deposits and relied on version vectors for the

[7] The dagger † indicates the replacement, in the indexed relation, of the active tuple to apply its incrementation.

detection of concurrent replicas. Upon detection of concurrency an upcall was made to the application so that it issued an appropriate merge that would superseed both.

Theoretical work on merging of software code[1] faces similar problems when modeling processes of combining changes to programs, however the changes are often incompatible and thus the model must handle reversal of changes which makes it substancially different from our reconciliation model.

6 Conclusions

Replication is used to increase availability in distributed systems where disconnection or network partitions are frequent. However, traditional replica control mechanisms such as quorum consensus, primary replicas and other strong consistency approaches are unable to provide a high level of availability on mobile environments. This papers describes an environment where mobility is rather unconstrained because creation, use and joining of replicas is done autonomously taking advantage of pair-wise communication. A formal technique based on the SETS Calculus was used to specify composable components that enjoy several properties which guarantee that pair-wise joining of replicas will lead to eventual consistency. This allows the construction of complex structures that also ensure convergence and shows the necessary properties that must be met by new components. Component specifications can be checked for compliance with the environment by relating them to the its rules, as is exemplified in the appendix.

This work proceeds in several fronts. On a practical side, several components, including those presented here, have been implemented on a Java hierarchy and an intermediate language was developed to give persistence to structured instances of components. A non-trivial sample reconciliator was built for Netscape bookmarks by composing six different component types, including Const, IncMap and ConstSeq and some new components. On a more formal level, an attempt is being made to describe the semantic dependencies of operations in the ADT's signature in order to establish a technique for choosing the appropriate state and order relation that conforms to our condition for unconstrained mobility.

References

[1] Valdis Berzins. Software merge: Semantics of combining changes to programs. *ACM Transactions on Programming Languages and Systems*, 16(6):1875–1903, November 1994.

[2] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[3] Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, and Brent Welch. The bayou architecture: Support for data sharing among mobile users. In *IEEE Workshop on Mobile Systems and Aplications*, Computer Science Laboratory, Xerox Palo Alto Research Center, December 1994.

[4] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Gerald J. Popek, and Dieter Rothmeier. Implementation of the ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[5] Silvano Maffeis, Walter Bischofberger, and Kai-Uwe Matzel. A generic multicast transport service to support disconnected operation. Technical report, Department of Computer Science, Cornell University, 1995.

[6] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *The 14th IEEE International Conference on Distributed Computer Systems*, pages 56–65, June 1994.

[7] J. N. Oliveira. Software reification using the sets calculus. In *BCS FACS 5th Refinement Workshop*, pages 141–171. Springer Verlag, January 1992.

[8] J. N. Oliveira. Sets - a data structuring calculus and its application to program development. Lecture Notes of Course at UNI/IIST in Macau, May 1997. Available at http://www.di.uminho.pt/~jno/ps/unu97set.ps.

[9] Peter Reiher, John Heidemann, David Ratner, Gregory Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *Proceedings of the Summer Usenix Conference*, 1994.

[10] Peter Reinher, Jerry Popek, Michial Gunter, John Salomone, and David Ratner. Peer-to-peer reconciliation based replication for mobile computers. In *Proceedings of ECOOP'96 II Workshop on Mobility and Replication*, July 1996. http://www.di.uminho.pt/~cbm/wmr96.html.

[11] M. Satyanaraynan, James Kistler, Puneet Kumar, Maria Okasaki, Ellen Siegel, and David Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, April 1990.

[12] Antonio Sousa, Carlos Baquero, Jose Orlando Pereira, Rui Oliveira, and Francisco Moura. A human centered perspective for mobile information sharing and delivery. In *Proceedings of ECOOP'96 II Workshop on Mobility and Replication*, July 1996. http://www.di.uminho.pt/~cbm/wmr96.html.

[13] Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in bayou a weakly connected replicated storage system. In *Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, Dec 1995. ACM.

A. Correctness of the IncSet component

Here we analyze the correctness of one component, the **IncSet**. Its specification is recalled bellow after integrating the inherited operations.

Type: IncSet
State:
$\Sigma = 2^X, I = Y$
Model:
$Init(\sigma', \iota')$
post $\sigma' = \oslash, \iota' = \perp$

$Insert(\sigma, e, \sigma')$
post $\sigma' = \sigma \cup \{e\}$

$Find(\sigma, e, b')$
post $\begin{cases} e \in \sigma \Rightarrow b' = true \\ e \notin \sigma \Rightarrow b' = false \end{cases}$

$Fork(\sigma, \iota, \sigma'_l, \iota'_l, \sigma'_r, \iota'_r)$
pre $\sigma \neq \perp$
post $\sigma'_l = \sigma'_r = \sigma, \iota'_l = \iota'_r = \iota$

$Join(\sigma_l, \iota_l, \sigma_r, \iota_r, \sigma', \iota')$
post $\sigma' = \sigma_l \cup \sigma_r$
$\iota' = \iota_l = \iota_r$

$Leq(\sigma_l, \sigma_r, b')$
post $\begin{cases} \sigma_l \subseteq \sigma_r \Rightarrow b' = true \\ else \Rightarrow b' = false \end{cases}$

Verifying the partial order

Our first step is to check that the operator *Leq* defines a partial order over the elements defined in the state. The state is specified as $\Sigma = 2^X$ and defines subsets of elements from a given set X. It is known [2] that for any set X, the powerset $\wp(X)$, consisting of all subsets of X, is ordered by set inclusion such that: for $a, b \in \wp(X)$, we define $a \leq b$ if and only if $a \subseteq b$. From the definition of *Leq* it is clear that the order is based on the relation $\subseteq$ over sets. The three properties that define the partial order: (reflexivity) $x \leq x$; (antisymmetry) $x \leq y \wedge y \leq x \Rightarrow x = y$; (transitivity) $x \leq y \wedge y \leq z \Rightarrow x \leq z$, are trivially respected by the $\subseteq$ relation on sets.

Checking operations against the Incorporation rule

Now that the order relation is established, we proceed to the analysis of the incorporation operations with respect to the **Incorporation** rule that relates them to the order relation.

The *Find* operation does not change the state (no new σ' is defined). All operations that do not change the state fall under the reflexivity property, if $a = b$ then $a \subseteq b$ implying

$a \leq b$, and thus fullfiling the **Incorporation** rule.

As for the *Insert* operation, it changes the state with the transformation $\sigma' = \sigma \cup \{e\}$. After the *Init* operation the state is $\sigma_0 = \oslash = \{\}$. The insertion of a element e, by the above transformation, derives a state $\sigma_1 = \{e\}$. Clearly $\{\} \subseteq \{e\}$ which implies that $\{\} \leq \{e\}$.

Subsequent insertions derive a state σ_{i+1} from a state σ_i by applying the same transformation and inserting a given element e. If $e \in \sigma_i$ then $\sigma_i \cup \{e\} = \sigma_i$ and thus $\sigma_i = \sigma_{i+1}$ that, again by reflexivity, leads to $\sigma_i \subseteq \sigma_{i+1} \Rightarrow \sigma_i \leq \sigma_{i+1}$. If $e \notin \sigma_i$ it is still trivial that $\sigma_i \subseteq \sigma_i \cup \{e\}$ which again leads to $\sigma_i \subseteq \sigma_{i+1} \Rightarrow \sigma_i \leq \sigma_{i+1}$.

Fork validation

The **IncSet** *Fork* operation was inherited from **Basic** and makes a simple copy of the state by the transformation $\sigma'_l = \sigma'_r = \sigma$. The **Fork** rule indicates that the resulting replicas are related by $\simeq$ to the original replica.

Consider the relation $a \simeq b$ expressed as $a \preceq b \wedge b \preceq a$ and then as $a \subseteq b \wedge b \subseteq a$. The above transformation indicates that $\sigma_r = \sigma$ and thus substituting both a and b by σ we obtain $\sigma \subseteq \sigma \wedge \sigma \subseteq \sigma$ which is a tautology. The same can be applied to σ_l, thus proving the **Fork** rule.

Join validation

Here the *Join* operation is defined by the transformation $\sigma' = \sigma_l \cup \sigma_r$ and must be validated under the **Join** rule. This rule states that when obtaining e_z by the *Join* operation over two replicas e_x and e_y, two properties must be met: (i) $e_x \preceq e_z$ and $e_y \preceq e_z$, (ii) $\forall e_i : e_x \preceq e_i$ and $e_y \preceq e_i$ implies that $e_z \preceq e_i$.

The first property is here translated as $e_x \subseteq e_z \wedge e_y \subseteq e_z$ and then, by applying the transformation, as $\sigma_l \subseteq \sigma_l \cup \sigma_r \wedge \sigma_r \subseteq \sigma_l \cup \sigma_r$ which is a simple tautology.

The second property, when translated into the $\subseteq$ relation, states that for any e_i such that $e_x \subseteq e_i \wedge e_y \subseteq e_i$ we must have $e_z \subseteq e_i$. By applying the transformation defined in the *Join* operation we obtain: $\forall \sigma_i : \sigma_l \subseteq \sigma_i \wedge \sigma_r \subseteq \sigma_i \Rightarrow \sigma_l \cup \sigma_r \subseteq \sigma_i$ which can be proved as follows.

From the sub-expression $\sigma_l \subseteq \sigma_i \wedge \sigma_r \subseteq \sigma_i$ we know that $x \in \sigma_l \Rightarrow x \in \sigma_i$ and that $y \in \sigma_r \Rightarrow y \in \sigma_i$. Considering an element z such that $z \in \sigma_l \cup \sigma_r$, we have $z \in \sigma_l \vee z \in \sigma_r$. Implying from above that $z \in \sigma_i \vee z \in \sigma_i$, so $z \in \sigma_i$. Which proves $z \in \sigma_l \cup \sigma_r \Rightarrow z \in \sigma_i$ and derives our goal $\sigma_l \cup \sigma_r \subseteq \sigma_i$.

Verification of the identifiers both in *Fork* and *Join* is not necessary since they do not interfere with the order relation expressed in *Leq*.