Verification and prototyping of distributed dataspace applications ⋆

Simona Orzan[1,2] and Jaco van de Pol[1]

[1] Centrum voor Wiskunde en Informatica,
P.O.Box 94079, 1090 GB, Amsterdam, The Netherlands
`simona,vdpol@cwi.nl`
[2] "A.I.Cuza" University, Iasi, Romania

Abstract. The *space calculus* is introduced as a language to model distributed dataspace systems, i.e. distributed applications that use a shared (but possibly distributed) dataspace to coordinate. The publish-subscribe and the global dataspace are particular instances of our model. We give the syntax and operational semantics of this language and provide tool support for functional and performance analysis of its expressions. Functional behaviour can be checked by an automatic translation to μCRL and the use of a model checker. Performance analysis can be done using an automatically generated distributed C prototype.

1 Introduction

A distributed system is generally seen as a number of single-threaded applications together with a distributed communication layer that coordinates them. Various shared dataspace models have been proposed to solve the task of coordination: Linda [8], Splice [3], JavaSpaces [14]), Gamma [1], WCL [21].

A shared dataspace (or tuple space) architecture is a distributed storage of information, viewed as an abstract global store, where applications read/write/delete pieces of data. In this paper, we focus on the problem of designing, verifying and prototyping distributed shared dataspace systems. Building correct distributed systems is a difficult task. Typical required properties include transparent data distribution and fault-tolerance (by application replication and data replication), which are usually ensured at the price of giving up some performance. Many questions occur when deciding on the exact shape of the distributed dataspace. For instance: what data should be replicated (in order to prevent single points of failure)? should the local storages be kept synchronized or should they be allowed to have different views on the global space? the migration of data between local storages should be on a subscription basis or rather "on demand"?

The space calculus, introduced in this paper, is an experimental framework in which verification and simulation techniques can be applied to the design of distributed systems that use a shared dataspace to coordinate their components. Tool support for the

space calculus can be used to analyze concrete systems, with respect to their functional behaviour, as well as their performance.

We model the shared space as a graph with atomic spaces as nodes (See Figure 1). We consider two types of links between spaces: *eager* and *lazy*. When elements are written in a local space, they are asynchronously transferred over all eager links that start in this local space. Eager links can be used to model subscription and notification mechanisms. Lazy links, on the other hand, are demand driven. Only when a data item is requested in some atomic space, it is transferred via a lazy link from one of the neigbouring spaces.
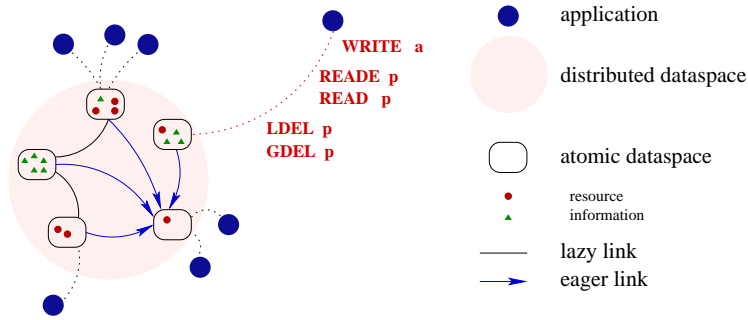

Fig. 1. A distributed dataspace architecture

Besides modelling the shared space, the space calculus provides a coordination language to represent the *applications*. This language contains various communication primitives, such as write, blocking and non-blocking read, local and global take operations, and static publish and subscribe marks. Applications are loosely coupled in the sense that they cannot directly address other applications. The precise behaviour of a system (space + applications) is given by an operational semantics.

We provide a tool that translates a space calculus specification into a μCRL specification [15]. From this code a state space graph can be generated and analyzed by means of the model checker CADP [13]. A second tool generates distributed C code to simulate the system. System designers may use the automatic verification and simulation possibilities provided by the μCRL toolset [2] to verify properties of their architecture. Complementary, the distributed C prototype can be used for testing purposes, and to get an indication about the performance (e.g. number of messages, used bandwidth, bottlenecks). Several design choices can be rapidly investigated in this way. Ultimately, the prototype C implementation could even be used as a starting point for building a production version.

The operational semantics provides the formal ground for algebraic reasoning on architectures. Despite its apparent simplicity, our calculus has powerful modelling capabilities. By restricting the allowed space-connectives and the allowed coordination primitives, we obtain well known instances, such as the kernels of Splice (a publish-subscribe architecture), and JavaSpaces (global space). Thus, the properties of various

dataspace architectures can be studied in our uniform framework. The verification tool will help getting fast insights in the replication and distribution behaviour of certain architectures, for instance. The simulation tool can help identifying the classes of applications appropriate to each architecture.

Related work. An overview of shared dataspace coordination models is given in [22]. Some work that studies different semantics has been done in [4, 5, 7, 6], on which we based the style of our operational semantics. [7] compares the publish/subscribe with the shared dataspace architectural style by giving a formal operational semantics to each of them. We also aim at being able to compare the two paradigms, but we take a more unifying perspective: we consider both as being particular instances of the more general distributed dataspace model and express them in the same framework.

[10] was the first attempt to use a Unity-like logic to reason on a shared dataspace coordination model (Swarm). [18] has similar goals. It provides a framework for describing software architectures in the theorem prover PVS. However, it seems that the verification of functional behaviour is out of the scope of that paper. In [9], a language for specification and reasoning (with TLA) about software architectures based on hyerarchical multiple spaces is presented. The focus there is on the design of the coordination infrastructure, rather than on the behaviour of systems using it. In [16], a translator from the design language VPL to distributed C++ code is presented. VPL specifications can be verified using the CWB-NC toolset. Compared to that approach, our work is more specific. We concentrate on shared dataspace architectures and define a "library" of carefully chosen set of primitives that are both handy and expressive. In [12], scenario-based verification is introduced as an useful technique in between verification and testing. Our language also supports that.

Organization of the paper. In section 2, we present the syntax and semantics of our calculus, we comment its main characteristics and we briefly introduce the tools that support the space calculus. Then (section 3) we try to give an idea, through some examples, about the applications of this framework. Finally, section 4 contains some concluding remarks.

2 The Space Calculus

2.1 Informal view

The most important goal of our work was to define a language that is both small and expressive. With so many existing shared dataspace models, it is difficult to decide what features are the most representative. Some of the choices that we are faced to are:
— atomic spaces can be sets or multisets;
— when transfering data items between different spaces, they could be replicated or moved;
— the primitives can be location-aware or location-independent
— the retrieve operation can be destructive or nondestructive
— global and local primitives
The answers depend of course on the specific application or on the purpose of the architecture. In order to allow the modelling of as many situations as possible, we choose

to let the user make the distinction between data items that should be treated as *information* (e.g. data from a sensor), for which multiplicity is not relevant, and data items that should be treated as *resource* (e.g. numbers to be added, jobs to be executed), for which multiplicity is very important. When handling elements, the space takes into account their type. The transfer between spaces means "copy" for information items and "move" for resources. Similarly, the lookups requested by applications are destructive for resources and non-destructive for information items.

The atomic spaces are multisets in which the elements tagged as information are allowed to randomly increase their multiplicity. As for the question whether to give to applications the possibility to directly address atomic spaces by using handles, like in [**?**], we have chosen not to, in order to keep the application layer and the coordination layer as separated as possible. The advantage of a clear separation is that the exact distribution of the space is transparent to the applications.

2.2 Syntax and semantics

As mentioned before, a system description consists in our view from a number of program applications and a number of connected atomic spaces. To bind the applications to the atomic spaces we use (abstract) locations, denoted by $i, j, ...$ The data items in the distributed space come from a set $\mathcal{D}$ of values, ranged over by $a, b, ...$ Furthermore, we assume a set of patterns $\mathcal{P}at(\mathcal{D})$, i.e. properties that describe subsets of $\mathcal{D}$. We assume that $\mathcal{D} \subseteq \mathcal{P}at(\mathcal{D})$. $p, q..$ denote patterns. We also postulate two predicates on patterns: $match : \mathcal{P}at(\mathcal{D}) \times \mathcal{D} \rightarrow \{\top, \bot\}$ to test if a given pattern matches a given value, and $inf : \mathcal{P}at(\mathcal{D}) \rightarrow \{\top, \bot\}$ to specify whether a given pattern should be treated as information or as resource. The predicate $res : \mathcal{P}at(\mathcal{D})$ will be used as the complementary of inf.

A program (**Proc**) is a program expression **P** residing at a location (i). A program expression is a sequence of coordination primitives. Formal parameters in programs are denoted by $x, y...$ ε denotes the empty program, and $\bot$ denotes a special error value.

$$Prim ::= write(a) \mid read(p, x) \mid read\exists(p, x) \mid ldel(p) \mid gdel(p)$$
$$P \quad ::= \varepsilon \mid Prim.P$$
$$Proc ::= [P]^i$$

The lazy and eager behaviours of the connections between atomic spaces are specified as special marks: $Pub^i_p, Sub^i_p, LL_{ij}$. If Pub^i_p and Sub^j_q are present, then all data matching $p \wedge q$ written in the space i by an application will be asynchronously forwarded to the space j. Thus, there is an eager link from i to j. The presence of LL_{ij} indicates that there is a (symmetric) lazy link from space i to j. That is, all data items of i are visible for retrieve operations addressed to j by an application.

For administrative reasons, the set of data items (a) is extended with buffered items that have to be sent ($!a^j$), pending request patterns ($?p$) and subscription policies ($\bigcirc p, k,$ $\bigcirc p, k, t$). Subscription policies are inspired by Splice and their function is to filter the data coming into a space as consequence of a subscription. Based on *keys* and *timestamps*, some of the data in the space will be replaced (overwritten) by the newly arrived element. The parameters k, t are functions on data $k : \mathcal{D} \rightarrow \mathcal{K}eys(D), t : \mathcal{D} \rightarrow \mathsf{N}$

that dictate how should the keys and the timestamps be extracted from data items. If the newly arrived element a, matching p, meets the filter $\bigcirc p, k$, then a will overwrite all data matching p with the key equal to that of a. If it meets the filter $\bigcirc p, k, t$, it will overwrite all data matching p with the key equal to that of a and timestamp not fresher than that of a. With this second filter, it is also possible that a drops out, if its timestamp is too big.

A configuration ($\mathcal{C}$) then consists of a number of atomic dataspaces and applications, each bound to a location, and a number of links:

$$
\begin{aligned}
data &::= a \mid !a^j \mid ?p \mid \bigcirc p, k \mid \bigcirc p, k, t \\
Data &::= \langle D \rangle^i, \text{ where } D \text{ is a finite set over } data \\
Link &::= \upharpoonright_i^j \mid \uparrow_p^i \mid \downarrow_p^i \\
Conf &::= Data \mid Proc \mid Link \mid Conf \parallel Conf
\end{aligned}
$$

The operational semantics of the space calculus is defined as a labeled transition relation $\mathcal{C} \xrightarrow{a} \mathcal{C}'$, meaning that if the system is in configuration $\mathcal{C}$ then it can do an a-step to configuration $\mathcal{C}'$. The transition relation is defined inductively in Figure 2. Note that the OS rules don't explicitely reflect the dual information/resource structure of the systems. This unitary appearance is possible due to a few operators on data, the definition of which (below) encapsulates this distinction. D, B are multisets, $-$ and $+$ denote the usual difference and union of multisets and d is a $data$ (a or $!a^j$ or $?p$ or $\bigcirc p, k$ or $\bigcirc p, k, t$). We will use the notation $inf(d)$ to express the value of the predicate inf for the pattern occuring in d. That is, $inf(!a^j) = inf(a)$ and $inf(?p) = inf(\bigcirc p, k) = inf(\bigcirc p, k, t) = inf(p)$. The same holds for res.

$$
\begin{aligned}
D \mathbin{\uplus_p} a &= D - [\, b \in D \mid match(p, b) \,] + [\, a \,] \\
D \mathbin{\uplus_{p,k}} a &= D - [\, b \in D \mid match(p, b) \wedge k(b) = k(a) \,] + [\, a \,] \\
D \mathbin{\uplus_{p,k,t}} a &= D - [\, b \in D \mid match(p, b) \wedge k(b) = k(a) \wedge t(b) \leq t(a) \,] + [\, a \,] \\
D \mathbin{\uplus_d} d &= D - [\, d \in D \,] + [\, d \,]
\end{aligned}
$$

$$
D \oplus d = \begin{cases} D \mathbin{\uplus_d} d & \text{if } inf(d) \\ D + [d] & \text{if } res(d) \end{cases} \qquad D \ominus a = \begin{cases} D & \text{if } inf(a) \\ D - [a] & \text{if } res(a) \end{cases}
$$

$$
D \oplus [\, d_1 \cdots d_n \,] = D \oplus d_1 \oplus \cdots \oplus d_n
$$

$$
D \uplus a = \begin{cases} D \mathbin{\uplus_{p,k}} a & \text{if } \bigcirc p, k \in D \wedge match(p, a) \\ D \mathbin{\uplus_{p,k,t}} a & \text{if } \bigcirc p, k, t \in D \wedge match(p, a) \\ D \oplus a & \text{if } \nexists \bigcirc p, k, \bigcirc p, k, t \in D \text{ s.t. } match(p, a) \end{cases}
$$

$$
B \boxplus d = \begin{cases} B \mathbin{\uplus_d} d & \text{if } inf(d) \\ [\, d \,] & \text{if } res(d) \end{cases}
$$

We now explain the intuitive semantics of the coordination primitives and how this semantics is modeled by the rules.

$write(a)$: write data item a into the local dataspace, to be automatically forwarded to all subscribed spaces. a is added to the local dataspace (W1) and an auxiliary $w(i, a)$

(W1) $\quad \langle D \rangle^i \; || \; [write(a).P]^i \xrightarrow{w(i,a,[a])} \langle D \rangle^i \; || \; [P]^i$

(W2) $\quad \dfrac{\mathcal{C} \; || \; \langle D \rangle^i \xrightarrow{w(i,a,B)} \mathcal{C}' \; || \; \langle D \rangle^i}{\mathcal{C} \; || \; \langle D \rangle^i \; ||\downarrow_p^i|| \uparrow_q^j \xrightarrow{w(i,a,B \boxplus !a^j)} \mathcal{C}' \; || \; \langle D \rangle^i \; ||\downarrow_p^i||\uparrow_q^j}$

$$match(p,a) \wedge p \cap q \neq \emptyset$$

(W3) $\quad \dfrac{\mathcal{C} \xrightarrow{w(i,a,B)} \mathcal{C}'}{\mathcal{C} \; || \; X \xrightarrow{w(i,a,B)} \mathcal{C}' \; || \; X}$

$$X \in \{|_j^k, [P]^j, \langle D \rangle^j, \downarrow_q^j, \uparrow_p^l\}, \quad p: \; not \; match(p,a) \vee \downarrow_p^i \notin \mathcal{C}$$

(W4) $\quad \dfrac{\mathcal{C} \xrightarrow{w(i,a,B)} \mathcal{C}'}{\mathcal{C} \xrightarrow{w(i,a,B \boxplus a)} \mathcal{C}'}$

(W5) $\quad \dfrac{\langle D \rangle^i \; || \; \mathcal{C} \xrightarrow{w(i,a,B)} \langle D \rangle^i \; || \; \mathcal{C}'}{\langle D \rangle^i \; || \; \mathcal{C} \xrightarrow{write(a)} \langle D \oplus B \rangle^i \; || \; \mathcal{C}'}$

(W6) $\quad \langle D + [!a^j] \rangle^i \; || \; \langle D' \rangle^j \xrightarrow{\tau} \langle D \rangle^i \; || \; \langle D' \uplus \{a\} \rangle^j$

(Rτ) $\quad \langle D \rangle^i \; || \; [read(p,x).P]^i \xrightarrow{\tau} \langle D + [?p] \rangle^i \; || \; [read(p,x).P]^i \qquad\qquad ?p \notin D$

(R) $\quad \langle D + [?p] \rangle^i \; || \; [read(p,x).P]^i \xrightarrow{read(p,a)} \langle D - [?p] \ominus a \rangle^i \; || \; [P[x := a]]^i$

$$a \in D \wedge match(p,a)$$

(R$\exists$1) $\quad \langle D \rangle^i \; || \; [read\exists(p,x).P]^i \xrightarrow{read\exists(p,a)} \langle D \ominus a \rangle^i \; || \; [P[x := a]]^i$

$$a \in D \wedge match(p,a)$$

(R$\exists$2) $\quad \langle D \rangle^i \; || \; [read\exists(p,x).P]^i \xrightarrow{read\exists(p,\perp)} \langle D \rangle^i \; || \; [P[x := \perp]]^i$

$$\nexists a \in D \; match(p,a)$$

(LD) $\quad \langle D \rangle^i \; || \; [ldel(p).P]^i \xrightarrow{ldel(p)} \langle D - [a \in D \; | \; match(p,a)] \rangle^i \; || \; [P]^i$

(GD1) $\quad [gdel(p).P]^i \; || \; ||_j \langle D_j \rangle^j \xrightarrow{gdel(p)} [P]^i \; || \; ||_j \langle D_j - [a \in D_j \; | \; match(p,a)] \rangle^j$

(GD2) $\quad \dfrac{\mathcal{C} \xrightarrow{gdel(p)} \mathcal{C}'}{\mathcal{C} \; || \; X \xrightarrow{gdel(p)} \mathcal{C}' \; || \; X} \qquad\qquad\qquad\qquad X \neq \langle D \rangle^i$

(TAU) $\quad \langle D + [?p] \rangle^i \; ||_i^j|| \; \langle D' \rangle^j \xrightarrow{\tau} \langle D - [?p] \oplus a \rangle^i \; ||_i^j|| \; \langle D' \ominus a \rangle^j$

$$a \in D' \wedge match(p,a)$$

(act) $\quad \dfrac{\mathcal{C} \xrightarrow{act} \mathcal{C}'}{\mathcal{C} \; || \; \mathcal{C}'' \xrightarrow{act} \mathcal{C}' \; || \; \mathcal{C}''} \qquad\qquad act \notin \{gdel(p), write(a), w(i,a)\}$

Fig. 2. Operational semantics of the space calculus

step is introduced. When pushing $w(i, a)$ to the top level, if a matches a pattern published by i, then $!a^j$ items are introduced for all subscriptions Sub_p^j matching a (rule W2, W3). At top level, the auxiliary $w(i, a)$ step can be promoted to a $write(a)$ step (W4). Finally, the a items are sent to the subscribed spaces asynchronously (W5). The operator $\uplus$ in the righthand side of rule (W5) states that the freshly received data item should be added to the local database taking into account the local subscription policies.

$read(p, x)$: blocking test for presence, in the local space and its lazy linked neigbouring spaces, of some item a matching p; x will be bound to a. This results in generating a $?p$ request, keeping the application blocked (Rτ). If a matching a has been found, it is returned and the application is unblocked (R). Meanwhile, the lazy linked neighbours of the local space asynchronously respond to the request $?p$, if they have an item matching p (TAU).

$read\exists(p, x)$: non-blocking test for presence in the local space. If some item a matching p exists in the local space, it is bound to x; otherwise a special error value $\perp$ is returned. Delivers a matching a from the local space, if it exists (R$\exists$1). Otherwise an error value is returned (R$\exists$2).

$ldel(p)$: local delete. This atomically removes all elements matching p from the local space (LD).

$gdel(p)$: this is the global remove primitive. It attomically deletes all items matching p, in all atomic spaces. Note that due to its global synchronous nature, $gdel$ can not be lifted over atomic spaces (GD2).

Finally, the general parallel rule (act) defines parallelism by interleaving, except for *write* and *gdel* which have their own parallel rules to ensure synchronization.

The kernels of some well-known dataspace paradigms can be obtained by restricting the allowed configurations and primitives.

Splice [3] implements a publish-subscribe paradigm. It has a loose semantics, reflecting the unstable nature of a distributed network. Applications announce themselves as publishers or subscribers of data sorts. Publishers may write data items to their local agents, which are automatically forwarded to the interested subscribers. Typically, the Splice primitives are optimized for real-time performance, and don't guarantee global consistency. The space calculus fragment without lazy links and restricted to the coordination primitives $write, read, ldel$ corresponds to the reliable kernel of Splice. Network searches (modeled by the lazy links) and global deletion (gdel) are typically absent. In Splice, data sorts have keys, and data elements with the same key may overwrite each other – namely at the subscriber's location, the "fresh" data overwrites the "old" one. The order is given by implicit timestamps that elements get in the moment when they are published. The overwriting is expressible in our calculus, by using the eager links with subscribe policies. The Splice's timestamps mechanism is not present, but some timestamping behaviour can be mimicked by explicitely writing and modifying an extra field in the tuples that models the data.

JavaSpaces [14] on the contrary can be viewed as a *global* dataspace. It typically has a centralized implementation, and provides a strongly consistent view to the applications, that can write, read, and take elements from the shared dataspace. The space calculus fragment restricted to a single atomic space to which all coordination primitives are attached, and with the primitives $write, read, read\exists$ forms a fragment of

JavaSpaces. Transactions and leasing are not dealt with in our model. Note that with the mechanism of marking the data "information" or "resource", we get the behaviour of both destructive and non-destructive JavaSpaces lookup primitives: our $read$, $readE$ works, when used for information, like $read$ and $readIfExists$ from JavaSpaces, and like $take$ and $takeIfExists$ when called for resources.

So, interesting parts of different shared dataspace models are expressible in this framework. A thorough study on how to improve this calculus so that it becomes a real comparison instrument remains to be done.

2.3 Verifying and prototyping

We defined a mapping from every configuration in the operational semantics to a process in the μCRL specification language [15]. A short description of this mapping is given in appendix A. We claim (proof pending) that any configuration of space calculus is weakly bisimilar to its μCRL image. The generation of the corresponding μCRL specification is automated. Therefore, the μCRL toolset [2] can be immediately used to simulate the behaviour of a configuration. This toolset is connected to the CADP [13] model checker, so that temporal properties on systems in the space calculus can be automatically verified by model checking. Typical verified properties are deadlock freeness, soundness, weak completeness, equivalence of different specifications.

Another tool translates space-calculus specifications to a distributed implementation in C that uses MPI (Message-Passing Interface) [**?**] for process communication. This automatically generated implementation has one process for each atomic space and one for each application. Different machines can be specified for different locations, thus getting a real distribution of spaces and applications. Typical performance measures are throughput, latency.

3   Applications of the space calculus

In order to make the space-calculus usable as specification language, the tools supporting the space calculus work with a more concrete syntax. The data universe considered is *tuples of naturals* and the patterns are incomplete tuples. Apart from the syntactical constructions already defined, we allow *assignment* of data variables, assignment of tuple variables and *if* and *while* with standard semantics. The examples below use this "tool language". A description of it, including a precise syntax written in a slightly simplified YACC format, is given in appendix B.

3.1   Towards distributed JavaSpaces

One of the initial motivations of our work was to model a distributed implementation of JavaSpaces, still providing the same strongly consistent view to the applications. This is the reason that we introduced the relatively expensive global delete. The expression

$$\langle \emptyset \rangle^0 \parallel \langle \emptyset \rangle^1 \parallel\downarrow_\star^0 \parallel\uparrow_\star^0 \parallel\downarrow_\star^1 \parallel\uparrow_\star^1$$

```
nfields = 1                          nfields = 1
upbound = 2                          upbound = 2
res <*>                             res <*>

space JS                            space JS
                                    space JSbis
app Ping@JS {                       JS -> <*>
  write <1>; EXTping; read <0> x;   JS <- <*>
  write <1>; EXTping; read <0> x;   JSbis -> <*>
}                                   JSbis <- <*>
app Pong@JS {
  read <1> x; write <0>; EXTpong;   app Ping@JS {
  read <1> x; write <0>; EXTpong;     write <1>; EXTping; read <0> x;
}                                     write <1>; EXTping; read <0> x;
                                    }
                                    app Pong@JSbis {
                                      read <1> x; write <0>; EXTpong;
                                      read <1> x; write <0>; EXTpong;
                                    }
```

Fig. 3. A Ping-Pong application on one Javaspace (left) and on two (right)

models a distributed JavaSpaces implementation, which should be robust against the crash of one space. Both spaces are eagerly linked by a subscription matching any item. In this context, distribution of the space should be completely transparent for the application processes.

Two rounds of the Ping-Pong game ([14], [20]) can be written in the space calculus as follows:

$$Ping = write(1).read(0, x).write(1).read(0, x)$$
$$Pong = read(1, x).write(0).read(1, x).write(0)$$

(with $\mathcal{D} = \{0, 1\}$ and $inf(x) = \bot, \forall x$). And we wish that they run on one Javaspace exactly the same as they run on two:

$$[Ping]^0 \parallel [Pong]^0 \parallel \langle\emptyset\rangle^0 = [Ping]^0 \parallel [Pong]^1 \parallel \langle\emptyset\rangle^0 \parallel \langle\emptyset\rangle^1 \parallel \downarrow_\star^0 \parallel \uparrow_\star^0 \parallel \downarrow_\star^1 \parallel \uparrow_\star^1$$

To check this equality, we only need to write the two specifications of the Ping-Pong game (with a single, respectively replicated space) in the "tool syntax" (Figure 3), generate the two statespaces and use the model checker to verify that they satisfy the *safety equivalence* relation.

3.2 Transparent replication of some Splice applications

Some of the most interesting problems in a system with components are associated with *replication*: what components can be replicated, and at what costs? We claim that the space calculus is a good framework for studying this type of questions. In the sequel we give an example of how our space calculus can be used to rapidly check transparent replication of some application components of Splice.

Consider a simple Splice system, composed of three applications: a *Producer* that writes data to the Splice network, based on observations that it makes on the environment; a *Transformer* that reads the data, applies some transformations on it and writes it back; and a *Consumer* that reads the transformed data items and uses it further, for instance by displaying it on a screen. The producer and the consumer are the components

```
nfields = 3                              nfields = 3
upbound = 3                              upbound = 3
space A1                                 space A1
space A2                                 space A2
space A3                                 space A3
A1 -> <1,*,*>                            space A4
A2 -> <2,*,*>                            A1 -> <1,*,*>
A2 <- <1,*,*> 1 3                        A2 -> <2,*,*>
A3 <- <2,*,*> 1 3                        A2 <- <1,*,*> 1 3
                                         A3 <- <2,*,*> 1 3
app Producer@A1                          A4 -> <2,*,*>
  {                                      A4 <- <1,*,*> 1 3
    itsp := 0;
    EXTin;                               app Producer@A1
    write <1,0,itsp>;                      {
    itsp := itsp + 1;                        itsp := 0;
    write <1,1,itsp>;                        EXTin;
  }                                          write <1,0,itsp>;
app Transformer@A2                           itsp := itsp + 1;
  {                                          write <1,1,itsp>;
    while (true) {                         }
      read <1,*,*> x;                   app Transformer@A2
      ivx := x/2+1;                       {
      itx := x/3;                           while (true) {
      write <2,ivx,itx>;                      read <1,*,*> x;
    };                                        ivx := x/2+1;
  }                                           itx := x/3;
app Consumer@A3 {                             write <2,ivx,itx>;
    while (true) {                          };
      read <2,*,*> x;                     }
      EXTout;                           app Transformer@A4
    };                                    {
  }                                         while (true) {
                                              read <1,*,*> x;
                                              ivx := x/2+1;
                                              itx := x/3;
                                              write <2,ivx,itx>;
                                            };
                                          }
                                        app Consumer@A3 {
                                            while (true) {
                                              read <2,*,*> x;
                                              EXTout;
                                            };
                                          }
```

Fig. 4. The Producer/Transformer/Consumer example

that interact with the environment, while the transformer works "under water". Therefore it is reasonable to ask whether it is possible to replicate the transformer without affecting the (external) behaviour of the system.

This producer-transformer-consumer example illustrates a specific pattern in Splice systems. The transparent replication of the middle component was extensively studied in [17], using both μCRL and PVS. We show how to model the problem in space-calculus (Figure 4), for the specific instance when two data items are produced, with values 0 and 1. The $itsp$ variable models the local clock. The two specifications are safety equivalent.

4 Conclusion

This paper presents our initial research in a unifying framework for various dataspace paradigms. We introduced the space calculus, in which various paradigms can be modeled. A formal syntax and operational semantics provides a rigorous basis to this calculus.

We aim at two goals: comparing the various paradigms with respect to their meta-properties and facilitating the analysis of individual systems based on shared dataspace architectures.

For the first goal, we view a particular dataspace paradigm as a fragment of the space calculus. Questions to be addressed are: does a fragment admit transparent replication of data/processes, what are the costs of a distributed implementation, what are the typical applications for certain fragments. An answer to the last question would facilitate early architectural design decisions. A number of these questions have been answered for Splice fragments already [11, 17, 19].

The second goal is supported by a translation to μCRL, of which we presented a fragment. A prototype automatic translator to μCRL exists. The resulting μCRL specifications can be used as an input to a model checker, thus formally establishing the functional correctness of a system. The approach follows a previous successful attempt for JavaSpaces [20].

Another tool generates distributed C-code from a system described in the space calculus. We have experimented with an implementation on top of the MPI-library (Message Passing Interface). The simulator thus obtained can be used to find performance bottlenecks in the high-level architecture. These could be solved by transforming the space calculus expression to a functionally equivalent one with a better performance.

As future work, we plan to investigate meta-properties for (fragments) of the space calculus and identify behaviour-preserving transformation rules. Also, we intend to study more examples, in order to establish the validity of our framework and to improve it where necessary. An interesting extension might be to allow dynamic creation of spaces and applications or the dynamic change of the inf predicate.

We like to acknowledge Michel Chaudron for initiating our quest for a unifying dataspace framework.

References

1. J.P. Banâtre and D. Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15:55–77, November 1990.
2. S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. Langevelde, B. Lisser, and J.C. van de Pol. μCRL: a toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. of 13th conference on Computer Aided Verification (CAV)*, pages 250–254, 2001. LNCS 2102.
3. M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, July 1993.
4. M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In J. Carroll, H. Haddad, D. Oppenheim, B. Bryant, and G.B. Lamont, editors, *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC '99)*, pages 146 – 155, San Antonio, Texas, USA, February 1999. ACM press.

5. N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In T. Rus, editor, *8th International Conference on Algebraic Methodology and Software Technology*, number 1816 in LNCS, Iowa, USA, 2000. Springer-Verlag.

6. N. Busi, C. Manfredini, A. Montresor, and G. Zavattaro. Towards a data-driven coordination infrastructure for peer-to-peer systems. In *Proc. of Workshop on Peer-to-Peer Computing*, number 2376 in LNCS. Springer-Verlag, May 2002.

7. N. Busi and G. Zavattaro. Publish/subscribe vs. shared dataspace coordination infrastructures. In *Proc. of WETICE'01*. IEEE Press, 2001.

8. N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course.* MIT Press, 1990.

9. P. Ciancarini, M. Mazza, and L. Pazzaglia. A logic for a coordination model with multiple spaces. *Science of Computer Programming*, 31(2–3):231–261, 1998.

10. H. Cunningham and G.-C. Roman. A unity-style programming logic for shared dataspace programs. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365–376, July 1990.

11. P.F.G. Dechering and E. de Jong. Transparent object replication: A formal model. In *Fifth Workshop on Object-oriented Real-Time Dependable Systems (WORDS'99F)*, Monterey, California, USA, 2000. IEEE Computer Society.

12. P.F.G. Dechering and I.A. van Langevelde. Towards automated verification of Splice in mu-CRL. Report SEN-R0015, CWI, Amsterdam, 2000.

13. J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP – a protocol validation and verification toolbox. In *Proc. 8th Conference on Computer-Aided Verification*, number 1102 in LNCS, pages 437–440. Springer, 1996.

14. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.

15. J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17. Elsevier, 2001.

16. D. Hansel, R. Cleaveland, and S. Smolka. Distributed prototyping from validated specifications. In *12th IEEE International Workshop on Rapid System Prototyping*, pages 97–102. IEEE Computer Society Press, June 2001.

17. J.M.M. Hooman and J.C. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings of SAC 2002 (Madrid)*, pages 351–358. ACM, 2002.

18. K. Lichtner, P. Alencar, and D. Cowan. A framework for software architecture verification. In *Proc. of 12th Australian Software Engineering Conference*, pages 149–158. IEEE Computer Society, 2000.

19. S.M. Orzan and J.C. van de Pol. Distribution of a simple shared dataspace architecture. In *Proc. of first Int. Workshop on Foundations of Coordination Languages, and Software Architectures*, Brno, 2002. To appear in ENTCS 68(3).

20. J.C. van de Pol and M. Valero Espada. Formal specification of JavaSpaces^{TM} architecture using μCRL. In F. Arbab and C. Talcott, editors, *Proc. of COORDINATION*, number 2315 in LNCS, pages 274–290. Springer, 2002.

21. Antony I. T. Rowstron. WCL: A co-ordination language for geographically distributed agents. *World Wide Web*, 1(3):167–179, 1998.

22. Robert Tolksdorf and Gregor Rojec-Goldmann. The SPACETUB models and framework. In *Coordination Models and Languages*, pages 348–363, 2002.

A The mapping to μCRL

The state of a μCRL system is the parallel composition of a number of processes. A process is built from atomic actions by sequential composition (.), choice (+,sum), conditionals (· ◁ · ▷ ·) and recursive definitions.

For our purpose, we introduce processes for each atomic space and for each application. An additional process, called the `TokenManager`, has to ensure that operations requiring global synchronization (*gdel*) don't block each other, thus don't introduce deadlocks. Before initiating a global delete operation, a space has to first request and get the token from the manager. When it has finished, it has to return the token to the manager, therefore allowing other spaces to execute their *gdel*s. A second additional process, `SubscriptionsManager`, manages the list (multiset) of current subscriptions. When an item *a* is written to an atomic space, that space synchronizes with the `SubscriptionsManager` in order to get the list of the other atomic spaces where the new item should be replicated or moved. For simplicity, we model the data items as

```
Atomic (id:Nat, D:TupleSet, Req: TupleSet,
        ToSend: NatTupleSet, todel:Tuple,
        LL: NatSet, PL: TupleSet, SL: SubscriptionList) =

%  W
  sum(v:Tuple,
      W(v). sum(NewToSend: NatTupleSet,
            sum(NewD: TupleSet,
                getToSend(v, ToSend, NewToSend, D, NewD).
                Atomic(id, NewD, Req, NewToSend,
                       todel, LL, PL, SL)))
      <| and(isData(v), match(v, PL)) |> delta)
+ sum(v:Tuple,
      W(v).
      Atomic(id, a(v,D), Req, ToSend, todel, LL, PL, SL)
      <| and(isData(v), not(match(v,PL)))|> delta )

% async send
+ sum(x:Nat, sum(y:Tuple,
      el_send(x,y).
      Atomic(id, D, Req, r(x,y,ToSend), todel, LL, PL, SL)
      <| in(x,y,ToSend) |> delta ))

% async receive
+ sum(x:Tuple,
      el_recv(id,x).
      Atomic(id, add(x,D,SL), Req, ToSend, todel, LL, PL, SL))
...
```

Fig. 5. Fragment from a μCRL specification of an atomic space

tuples of natural numbers – fields are modelled by the μCRL datasort `Nat`, tuples by `Tuple`. A short overview of the most important datasorts:

- `Bool` - true, false
- `Nat` - models the ids of atomic spaces and the fields of the tuples
- `Natset` - models the set of lazy links of a space
- `Tuple` - models the data items (a) and the patterns (p)
- `TupleSet` - models the as in an atomic space, the set of requests ($?ps$), and the set of published patterns.
- `NattupleSet` - is a set of pairs (Nat, Tuple); models sets of $!a^i$
- `SubscriptionList` - models the $\bigcirc p, k, \bigcirc p, k, t$

In Figure **??**, the μCRL specification of the space's $write$ behaviour is shown. An atomic space has two interfaces: one to the application processes, and one to the other atomic spaces. In μCRL calls between processes are modeled as synchronization between atomic actions. The primitives of the space calculus correspond to the following atomic actions of `Atomic`: $\{W, R, RE, Ldel, Togdel, Gdel\}$. The interface to the other atomic processes is used to send/receive data items and patterns for read requests.

The application programs are also mapped to μCRL processes. Execution of coordination primitives is modeled as atomic actions, that synchronize with the corresponding local space's pair actions. The synchronization with the space is described by the following communication function (the left column contains the actions of the space, the right one those of the application):

$$
\begin{array}{llll}
W & |\ write\ = w & Ldel & |\ ldel\ = ld \\
R & |\ read\ = r & Togdel & |\ togdel\ = \tau \\
RE & |\ readE = re & Toread & |\ toread = \tau \\
Gdel & |\ gdel\ = gd
\end{array}
$$

B The space-calculus tool language

Since we allow exactly one space per location, it is nice to give *names* to spaces and to say, instead of saying that a program stays at location i, that the program runs at the space <name>. A specification of a configuration consists of:

- (optional) fixing the tuple size (**nfields**) and the first natural value strictly greater than any field of any tuple(**upbound**). The default values are nfields=1, upbound=2.
- (optional) define the inf/res predicates, by mentioning the patterns for which res should be $\top$. Any pattern p not included by the **res** declaration has $inf(p) = \top$. - describing the spaces, by giving each space a name and, optionally, the machine where it's supposed to live. The default machine is "localhost".
- describing the applications, by specifing for each application its name, the name of the space with which it shares the location (the physical location as well) and its program.

Apart from the primitives **read**, **readE**, **write**, **ldel**, **gdel**, the actual language includes some extra constructions to provide easy data manipulation and control possibilities: natural variable names and expressions, projection of a tuple on a field, assignments, if, while, external actions that can be specified as strings.

```
EXTCOMMAND means [E][X][T][a − zA − Z]+
INTID means [i][a − zA − Z0 − 9]⋆
ID means [a − zA − Z][a − zA − Z0 − 9]⋆ (that is not INTID)
INT means [0 − 9]+


configuration      : settings declarations
settings           :
                   | setting settings

setting            : nfields = INT
                   | upbound = INT
                   | res pattern

declarations       : space declarations
                   | link declarations
                   | application declarations

space              : space ID ( ID )
                   | space ID

link               : LL ( ID , ID )
                   | ID − > pattern
                   | ID < − pattern | ID < − pattern intlist | ID < − pattern intlist INT

pattern            : < tuple >

tuple              : datum
                   | tuple , datum
datum              : * | INT | INTID
intlist            : INT | intlist , INT
intexpression      : INT | INTID | projection | intexpression + intexpression
projection         : pattern / INTID | ID / INTID
application        : app ID @ ID { program }

program            :
                   | command ; program

command            : write pattern
                   | write ID
                   | read pattern ID
                   | readE pattern ID
                   | ID := pattern
                   | INTID := intexpression
                   | ldel pattern
                   | gdel pattern
                   | publish pattern
                   | subscribe pattern | subscribe pattern intlist | subscribe pattern intlist INT
                   | if condition { program }
                   | while condition { program }
                   | EXTCOMMAND

condition          : ID | not(ID) | true | false
```

Fig. 6. The YACC style syntax definition

The condition of *if* and *while* is very simple: a standard boolean value or a variable name that gets tested for correctness. Namely, "*if* x" means "if x is not *error*". Extending the conditions is further work.