A Formal Z Specification of ProSet-Linda*

W. Hasselbring

University of Essen
Fachbereich Mathematik und Informatik — Software Engineering
Schützenbahn 70, 4300 Essen 1, Germany
willi@informatik.uni-essen.de

September 24, 1992

Abstract

ProSet is a set-oriented prototyping language. The coordination language Linda provides a distributed shared memory model, called tuple space, together with some atomic operations on this shared data space. In ProSet-Linda the concept for process creation via Multilisp's futures is adapted to set-oriented programming and combined with Linda's concept for synchronization and communication via tuple space. This paper presents a formal specification of the semantics of this combination by means of the formal specification language Z.

*This is Informatik-Bericht 04.92, University of Essen

Contents

1 **Introduction** 1

2 **The formal semantics of ProSet-Linda** 2

2.1 Basic definitions . . . 3

2.1.1 Abstractions for the embedding into the computation language . . . 3

2.1.2 Types and values . . . 3

2.1.3 Tuples . . . 5

2.1.4 Formals and templates . . . 6

2.2 Matching . . . 7

2.3 Tuple spaces . . . 8

2.4 Programs and processes . . . 9

2.5 Multiple tuple spaces . . . 11

2.5.1 CreateTS . . . 12

2.5.2 ExistsTS . . . 13

2.5.3 ClearTS . . . 14

2.5.4 RemoveTS . . . 15

2.6 Tuple-space operations . . . 15

2.6.1 Some preliminary definitions . . . 17

2.6.2 Depositing tuples . . . 18

2.6.3 Fetching tuples . . . 22

2.6.4 Meeting tuples . . . 26

2.7 Execution . . . 27

2.8 Fairness . . . 28

3 **Correctness of the design for an implementation** 29

3.1 Semantics vs. implementation design . . . 30

3.2 Optimizations . . . 30

3.3 Finite computer systems . . . 31

3.4 Fairness . . . 31

4 **Conclusions** 32

A **The prototyping language ProSet** 34

A.1 Data structures . . . 34

A.2 Control structures . . . 34

A.3 An example . . . 35

B The specification language Z **37**

B.1 Schemas 37

B.2 Axiomatic descriptions 38

B.3 Generic descriptions 39

B.4 Free type definitions 39

B.5 Expressions 40

C The coordination language Linda **41**

D The informal semantics of ProSet-Linda **43**

D.1 Process creation 43

D.1.1 Multilisp's futures 43

D.1.2 Process creation in ProSet 43

D.1.3 Program and process termination 44

D.2 Tuple-space operations 45

D.2.1 Depositing tuples 45

D.2.2 Fetching tuples 46

D.2.3 Meeting tuples 48

D.2.4 Nondeterminism and fairness while matching 49

D.3 Multiple tuple spaces 50

E Types of all names defined globally **51**

References **56**

Index of formal definitions **58**

Index of explained standard Z symbols and keywords **60**

1 Introduction

The definition of Linda has been presented informally [Gelernter, 1985] and, as a result, has included several ambiguities. E.g. [Narem, 1989] summarizes four basic types of process creation used in implementations of C-Linda's `eval` operation. These are different interpretations of the informal specification of the `eval` operation. Additional discussions of problems with the semantics of the `eval` operation may also be found in [Leichter, 1989] and in [Hasselbring, 1991]. Such a situation demands a more precise definition. However, informal descriptions are very valuable because it is easy to grasp the gist of the semantics without much effort. The popularity of Linda can in part be ascribed to this property.

Informal descriptions are in general accessible and intelligible to a wide community, but have the disadvantage of being prone to omissions and ambiguities as it was the case with Linda. Much recent work has been devoted to finding more rigorous mathematical methodologies which — though less accessible — are complete and unambiguous. One methodology is denotational semantics: in this approach each program phrase is given a denotation, or a meaning, as an object of some mathematical domain. It is compositional in the sense that the meaning of each phrase is a function of the meaning of its subphrases. A second methodology is operational semantics: in this approach rules are given for the evaluation of each phrase. Operational semantics is also compositional since the evaluation of each phrase is defined in terms of the evaluation of its subphrases. An operational semantics has the advantage of suggesting a possible implementation, and of easing comparisons with other languages having analogous formal semantics.

Formalizations of the coordination language Linda have already been undertaken with structured operational rules such as Plotkin's Structural Operational Semantics (SOS) or Milner's Calculus of Communicating Systems (CCS), with Petri Nets, the Chemical Abstract Machine, Term Rewriting Systems, Communicating Horn Clause Logic, Algebraic Specifications, and the formal specification language Z:

SOS, CCS	[Jensen, 1990; Jensen, 1992; Callsen *et al.*, 1991]
	[Hazelhurst, 1990; Ciancarini *et al.*, 1992]
Petri Net	[Ciancarini *et al.*, 1992]
Chemical Abstract Machine	[Ciancarini *et al.*, 1992]
Term Rewriting System	[Jagannathan, 1990] (for Scheme-Linda)
Communicating Horn Clause Logic	[Bosschere and Wulteputte, 1991] (for Multi-Prolog)
Algebraic Specification	[Anderson *et al.*, 1990] (for TS-Prolog)
Z	[Butcher, 1991]

A comparative study of some approaches may be found in [Ciancarini *et al.*, 1992]. In this paper we will present the operational semantics of tuple spaces in ProSet by means of the formal specification language Z. We consider the state of a program as the state of a set of tuple spaces and a set of active processes, and define the effect of tuple-space operations on the state. To obtain a correct implementation, the semantics of an implementation design for ProSet-Linda then has to be the same with respect to this program state.

This paper is organized as follows: the next section presents the formal semantics of ProSet-Linda. The appendices provide short introductions to the basic concepts: appendix A to the prototyping language ProSet, appendix B to the specification language Z, appendix C to the coordination language Linda, and appendix D to the informal semantics of ProSet-Linda. The reader may refer to these appendices according to his or her preliminary knowledge before reading further. Appendix E and the index of formal definitions at the end of this document provide a summary of the present formal specification. Section 3 discusses the correctness of the design for an implementation under the presented semantics and section 4 draws some conclusions.

We shall only indicate exception handling in the present specification. We refer to [Doberkat *et al.*, 1992] for an informal specification of exception handling in PROSET.

The discussion on formal semantics of Linda with Keld Kondrup Jensen and on the specification language Z with Ana Lucia Cavalcanti as well as the comments on drafts of this paper by Ernst-Erich Doberkat and Stephen Gilmore were very helpful.

2 The formal semantics of PROSET-Linda

The formal specification language Z [Spivey, 1992b] was chosen as a means for presenting the formal semantics of PROSET-Linda for several reasons. Firstly, Z has many similarities with PROSET: both languages are based on set theory and the predicate calculus. This alleviates the access to the formal specification for readers who are familiar with PROSET. Secondly, there are some tools available to support the construction of specifications in Z [Parker, 1991]. However, there exist some significant differences between PROSET and Z:

- PROSET is weakly typed, whereas Z is strongly typed.
- PROSET programs are executable prototypes, whereas Z specifications are not executable.
- PROSET only supports finite sets, whereas Z also supports infinite sets.

Z has no features for specifying parallelism. However, this does not prevent us from specifying a parallel programming language with Z: we shall model concurrency by an arbitrary interleaving of a set of atomic transactions performed by the acting processes. The goal of the presented work is not to specify as much parallelism as possible. The goal is to provide a precise specification of the semantics of generative communication in PROSET.

A Z specification consists of a combination of a formal text and a natural language description. The formal text provides the precise specification while the natural language text introduces and explains the formal parts. The formal text has two parts: the schema language, which provides a means of structuring the specification, and the mathematical language, which allows for the preciseness of the specification. The mathematical language is based largely on set theory and enables an abstract mathematical view of the objects being specified to be taken. The schema language enables specifications of large systems to be broken into more manageable sections.

The combination of natural language for explanation, and of the schema language for structurization produces specifications that are more readable than only mathematical formulas. In addition, the mathematical nature of the specifications enables implementors to use mathematical proofs to ensure the correspondence of their implementations with the specification.

Besides the clear advantage of writing the semantics in a mechanically checkable formalism, a formal specification discloses subtleties as well as difficulties that are otherwise *swept under* the carpet of an imprecise notation. The formal specification emerges as a contract — stating rights and obligations — between language designer and implementor, and it is an abstract, detailed language manual for the programmer.

The present presentation is meant to be self-contained and no previous knowledge of Z is required to understand it — at least we hope so. Where necessary, we provide notes to explain the notation used (enclosed in the symbols Z and Z). The index of explained Z symbols and keywords at the end of this document refers to these explanations. Readers familiar with Z may skip both, appendix B and the notes on Z. The presented specification has been developed with the *f*UZZ package [Spivey, 1992a].

Notational convention: components of PROSET programs are displayed in `typewriter` font to set them apart from Z specifications, which are displayed in *slanted* font.

2.1 Basic definitions

Appendix E provides a summary of all names defined globally in the specification with their associated types. Together with the index of formal definitions at the end of this document this should provide a comprehensive overview of our formal specification. The basic, given types are *Expression*, *LValue*, *Process*, *Statement*, and *Value*. Some additional basic types are based in these types and introduced via free type definitions.

2.1.1 Abstractions for the embedding into the computation language

The aim of this work is not to provide a formal semantics for the entire ProSet language. We restrict ourselves to specifying generative communication. However, for the embedding in the computation part we need connections to some basic concepts of ProSet. At first we need basic types for describing l-values and unevaluated expressions:

$$[Expression, LValue]$$

For a detailed discussion of expressions and l-values, and their relationship in ProSet we refer to [Doberkat *et al.*, 1992]. Note that l-values are not typed. Additional necessary basic types are statements and processes:

$$[Statement, Process]$$

Each process is unique. New processes may be spawned and existing ones may terminate. Note that processes have no first-class rights in ProSet. We shall need a notion for execution of statements:

$$Execute\ _ : \mathbb{P}\ Statement$$
$$\forall\, s : Statement \bullet Execute\ s$$

Z The underscore _ indicates the position of operands thus *Execute* is an unary predicate. A predicate is identified with the set of objects for which the predicate holds. $\mathbb{P}$ yields the power set of its operand. Z

Statements never yield values in ProSet.

2.1.2 Types and values

We have to know a few specific things about types and values in our specification. The unary operator **type** yields a predefined type-atom according to the type of its operand (see appendix A). The following equations hold in ProSet:

```
type 1 = integer
type integer = atom
type type type 1 = atom
```

No particular basic type for the type-names **atom**, **boolean**, **integer**, **real**, **string**, **tuple**, **set**, **function**, **modtype**, and **instance** is needed in our Z specification. ProSet does not employ the type matching known from C-Linda and similar embeddings of Linda into statically typed languages. Instead, *conditional value matching* as described in sections D.2.2 and 2.2 is employed. It is sufficient to model types in ProSet through the **type** operator and the predefined type-atoms, which are values:

$$[Value]$$

$$
\begin{array}{|l}
atom, boolean, integer, real, string, tuple, set, function, modtype, instance : Value \\
TRUE, FALSE : Value \\
om : Value \\
ValuesOfType : Value \rightarrowtail\!\!\!\!\!\!\!+\; \mathbb{P}\, Value \\
\hline
\mathrm{dom}\ ValuesOfType = \\
\quad \{atom, boolean, integer, real, string, tuple, set, function, modtype, instance\} \\
\mathrm{dom}\ ValuesOfType \subset ValuesOfType\ atom \\
ValuesOfType\ boolean = \{TRUE, FALSE\} \\
(\{ om \mapsto \{om\} \} \cup \{ t : \mathrm{dom}\ ValuesOfType \bullet t \mapsto ValuesOfType\ t \})\ \mathsf{partition}\ Value
\end{array}
$$

Z $X \rightarrowtail\!\!\!\!\!\!\!+\; Y$ is the set of partial injections from X to Y. dom yields the domain of a relation. A set S is a *proper subset* of a set T ($S \subset T$) if every member of S is also a member of T and if in addition S is different from T. The subset relation symbol is $S \subseteq T$. The notation $x \mapsto y$ is a graphical way of expressing the ordered pair (x, y). Z

Every value in ProSet, except for `om`, belongs to exactly one type set. The Boolean values are `true` and `false` as usual. We use capital letters for *TRUE* and *FALSE* in our specification, because *true* and *false* are predefined in Z. The undefined value `om` has no type. Applying the unary operator `type` to `om` is undefined, and thus yields the undefined value (`type om = om`). The corresponding function is *Type*:

$$
\begin{array}{|l}
Type : Value \longrightarrow Value \\
\hline
atom = Type\ atom = Type\ boolean = Type\ integer = Type\ real = Type\ string = \\
\qquad Type\ tuple = Type\ set = Type\ function = Type\ modtype = Type\ instance \\
boolean = Type\ TRUE = Type\ FALSE \\
om = Type\ om \\
\forall x : Value \mid x \neq om \bullet \\
\quad x \in ValuesOfType\,(Type\ x)
\end{array}
$$

Z $X \longrightarrow Y$ is the set of total functions from X to Y. The predicate $a = b = c$ is an abbreviation for $a = b \wedge b = c$. Z

For our purposes it is not necessary to specify the types of ProSet through an additional given, basic type. It is sufficient to specify the semantics of the `type` operator. The remainder of our specification would not change if we remove or add some type names (except for *atom*, *boolean*, *integer*, and *tuple*).

Notions for evaluation of expressions and the return values of processes shall be needed:

$$
\begin{array}{|l}
Evaluate : Expression \rightarrow\!\!\!\!\!+\; Value \\
ProcRetVal : Process \rightarrow\!\!\!\!\!+\; Value
\end{array}
$$

Z $X \rightarrow\!\!\!\!\!+\; Y$ is the set of partial functions from X to Y. Z

These are partial functions because the evaluation of expressions and processes might not terminate, and thus not produce a result. We also need a notion for assignment of values to *l*-values:

$$
\begin{array}{|l}
_\ \mathsf{IsAssigned}\ _ : LValue \longleftrightarrow Value
\end{array}
$$

Z $X \longleftrightarrow Y$ is the set of binary relations between X and Y. Z

For every pair (*lhs*, *rhs*), which is related by IsAssigned, an assignment of the value of *rhs* to *lhs* is modeled. We do not need a more detailed specification of assignment in our specification. We display identifiers in sans serif font when we use them as infix relation or operation symbols.

2.1.3 Tuples

Tuples in ProSet have their mathematical semantics as ordered sequences of objects; a value may appear multiply in a tuple, the order of components appearing in the tuple is relevant. Tuple components may be passive values or executing processes in our specification:

$$TupleComp ::= TupleValue \langle\langle Value \rangle\rangle \mid TupleProcess \langle\langle Process \rangle\rangle$$

Conceptually a tuple in ProSet is an infinite vector with almost all components equal to the undefined value `om`. The indexing of tuple components starts with the index `1`, the length returned by the `#` operator of ProSet is the largest index of a component different from `om`, thus `#[1,om] = 1` and `#[om,1] = 2` hold. Since almost all components in a tuple are equal to the undefined value `om`, we are able to specify active and passive tuples via finite sequences:

$$APTuple == \mathrm{seq}\ TupleComp$$

Z seq X is the set of finite sequences over X. These are finite functions from $\mathbb{N}$ to X whose domain is a segment $1 \mathrel{..} n$ for some natural number $n \in \mathbb{N}_1$. $\mathbb{N}_1 = \mathbb{N} \setminus \{0\}$, where $\setminus$ is the set difference operation. If a and b are integers, $a \mathrel{..} b$ is the set of integers between a and b inclusive. If $a > b$ then $a \mathrel{..} b$ is empty, thus sequences may be empty. Z

Note that only passive tuples are first-class objects in ProSet. However, in our specification for the basic type $APTuple$ we do not distinguish between passive and active tuples for simplicity. As we shall see in section 2.2, the matching procedure will distinguish between passive and active tuples. There exists a partial function between tuples in tuple space and passive tuples in ProSet:

$$\begin{array}{|l}
APTupleToValue : APTuple \rightarrowtail\!\!\!\!\!\!\;+\; Value \\
\hline
\forall\, tup : APTuple \mid \mathrm{ran}\ tup \subseteq \mathrm{ran}\ TupleValue \bullet \\
\qquad Type\ (APTupleToValue\ tup) = tuple
\end{array}$$

We shall not use $APTupleToValue$ in the remainder of our specification. Our intention to provide it is to show that there exists a partial function from active/passive tuples in our specification to passive tuple values in ProSet.

Note also that the following equations hold in ProSet:

```
    [om] = []
  [1,om] = [1]
  [om,1] /= [1]
```

The predefined function # for finite sets in Z will not work well to specify the unary `#` operator for tuples in ProSet, because the `#` operator yields the largest index of a component different from `om`:

```
#[om] = 0
```

$$\#\langle om \rangle = 1$$

We define the generic function $Arity$ instead:

$$\begin{array}{|l}
[X] \\
\hline
Arity : (Value \rightarrowtail X) \longrightarrow \mathrm{seq}\ X \longrightarrow \mathbb{N} \\
\hline
\forall\, tup : \mathrm{seq}\ X;\ Xvalue : Value \rightarrowtail X \bullet \\
\qquad Arity\ Xvalue\ tup = max(\{0\} \cup \{ i : \mathbb{N} \mid tup\ i \neq Xvalue\ om \})
\end{array}$$

Z max yields the maximum of a set of integers. Note that function application associates to the left in Z, so $f\ x\ y$ means $(f\ x)\ y$. Z

Arity is generic, because it applies to tuples in the same way as it shall apply to templates. *TupArity* then works for tuples:

$$TupArity : APTuple \longrightarrow \mathbb{N}$$
$$TupArity = Arity\ TupleValue$$

TupArity yields the largest index of an *APTuple* component which is different from the *TupleValue* of *om*.

2.1.4 Formals and templates

To model formals we need auxiliary type definitions for optional *l*-values and optional `into` expressions:

$$OptLValue ::= NoLValue \mid IsLValue\langle\langle LValue\rangle\rangle$$
$$OptInto ::= NoInto \mid IsInto\langle\langle Expression\rangle\rangle$$

Formals then consist of optional *l*-values and optional `into` expressions:

Formal
$$Destination : OptLValue$$
$$Into : OptInto$$

Components of templates are formals and values:

$$TempComp ::= TempValue\langle\langle Value\rangle\rangle \mid TempFormal\langle\langle Formal\rangle\rangle$$

Templates consist of a sequence of template components and a conditional expression:

Template
$$List : \mathrm{seq}\ TempComp$$
$$Condition : Expression$$

If there is no condition specified in a template, the ProSet-expression `true` is assumed. A notion for assignment of tuple components to formals of templates shall be needed:

$$_\ \mathsf{FormalAssign}\ _ : Template \leftrightarrow APTuple$$

$$\begin{array}{l}
\mathbf{let}\ FormalOf == TempFormal^{-1};\\
\quad ValueOf == TupleValue^{-1};\\
\quad LValueOf == IsLValue^{-1} \bullet\\
\forall\, temp : Template;\ tup : APTuple \mid \#temp.List = \#tup \wedge \mathrm{ran}\ tup \subseteq \mathrm{ran}\ TupleValue \bullet\\
\quad temp\ \mathsf{FormalAssign}\ tup \Leftrightarrow\\
\qquad (\forall\, i : \mathrm{dom}\ temp.List \mid temp.List(i) \in \mathrm{ran}\ TempFormal \wedge\\
\qquad\qquad (FormalOf(temp.List\ i)).Destination \in \mathrm{ran}\ IsLValue \bullet\\
\qquad\quad LValueOf((FormalOf(temp.List\ i)).Destination)\ \mathsf{IsAssigned}\ ValueOf(tup\ i))
\end{array}$$

Z **let** introduces local definitions for a predicate or for an expression. R^{-1} is the relational inverse of the relation R. $S.C$ is the notation for selecting a component C from a binding of a schema S. Z

For every pair $(temp, tup)$ which is related by FormalAssign an optional assignment to the *Destinations* of *temp* is modeled. A notion for evaluation of `into` expressions shall be needed:

$$
\begin{array}{l}
_\ \mathsf{EvalIntos}\ _ : (Template \times APTuple) \longrightarrow APTuple \\
\hline
\textbf{let}\ FormalOf == TempFormal^{-1}; \\
\qquad ExprOf == IsInto^{-1} \bullet \\
\forall\, temp : Template;\ tup, newtup : APTuple \mid \#temp.List = \#tup \land \mathrm{ran}\ tup \subseteq \mathrm{ran}\ TupleValue \bullet \\
\qquad (temp\ \mathsf{EvalIntos}\ tup = newtup) \Leftrightarrow \\
\qquad\qquad \#tup = \#newtup \land \\
\qquad\qquad (\forall\, i : \mathrm{dom}\ temp.List \mid temp.List(i) \in \mathrm{ran}\ TempValue \bullet \\
\qquad\qquad\qquad newtup\ i = tup\ i) \land \\
\qquad\qquad (\forall\, i : \mathrm{dom}\ temp.List \mid temp.List(i) \in \mathrm{ran}\ TempFormal \bullet \\
\qquad\qquad\qquad newtup\ i = \textbf{if}\ (FormalOf(temp.List\ i)).Into = NoInto \\
\qquad\qquad\qquad\qquad\qquad \textbf{then}\ tup\ i \\
\qquad\qquad\qquad\qquad\qquad \textbf{else} \\
\qquad\qquad\qquad\qquad TupleValue(Evaluate\ (ExprOf((FormalOf(temp.List\ i)).Into))))
\end{array}
$$

Z $X \times Y$ is the Cartesian product of X and Y (a set of pairs). Z

EvalIntos yields from a pair $(temp, tup)$ a new $APTuple$ that is equal to tup except for the fields in which $temp$ has `into` expressions. Those fields are replaced by the corresponding evaluated expression values. We shall need the function $TempArity$ for matching:

$$
\begin{array}{l}
TempArity : \mathrm{seq}\ TempComp \longrightarrow \mathbb{N} \\
\hline
TempArity = Arity\ TempValue
\end{array}
$$

It applies to template lists in the same way as $TupArity$ applies to tuples.

2.2 Matching

ProSet employs *conditional value matching* as informally specified in section D.2.2. A tuple and a template match, iff all the following conditions hold:

- The tuple is passive.
- The arities are equal.
- Values of actuals in templates are equal to the corresponding tuple fields.
- The Boolean expression behind | in the template evaluates to `true`. If no such expression is specified, then `|true` is the default.

As a first step we define matching of individual tuple and template components:

$$
\begin{array}{l}
_\ \mathsf{CompMatches}\ _ : TupleComp \leftrightarrow TempComp \\
\hline
\textbf{let}\ ValueOfTup == TupleValue^{-1}; \\
\qquad ValueOfTemp == TempValue^{-1} \bullet \\
\forall\, tupc : TupleComp;\ tempc : TempComp \bullet \\
\qquad tupc\ \mathsf{CompMatches}\ tempc \Leftrightarrow \\
\qquad\qquad tupc \in \mathrm{ran}\ TupleValue \land \\
\qquad\qquad (tempc \in \mathrm{ran}\ TempValue \Rightarrow ValueOfTup\ tupc = ValueOfTemp\ tempc)
\end{array}
$$

Therefore, only passive tuple components can match and, if the template component is **not** a formal, the *Value*s in the domain of the corresponding components have to be equal. Only passive tuples are relevant when considering tuple matching; active tuples are invisible to processes. Tuples and templates then match if their arities are equal, their corresponding components match, and the template condition holds. The template condition must yield a Boolean value:

$$
\begin{array}{|l}
\,\mathsf{Matches}\, : \mathit{APTuple} \leftrightarrow \mathit{Template} \\
\hline
\forall\, tup : \mathit{APTuple};\ temp : \mathit{Template} \bullet \\
\quad \mathit{Type}\,(\mathit{Evaluate}\ temp.\mathit{Condition}) = boolean \land \\
\quad tup\ \mathsf{Matches}\ temp \Leftrightarrow \\
\qquad \mathit{TupArity}(tup) = \mathit{TempArity}(temp.\mathit{List}) \land \\
\qquad \mathit{TRUE} = \mathit{Evaluate}\ temp.\mathit{Condition} \land \\
\qquad (\forall\, i : 1 \mathrel{..} \mathit{TupArity}(tup) \bullet tup(i)\ \mathsf{CompMatches}\ temp.\mathit{List}(i))
\end{array}
$$

The exception `type_mismatch` will be raised if the template condition does not yield a Boolean value. This is left in the formal specification.

2.3 Tuple spaces

Tuple spaces in our formal specification consist of an identity, its specified limit, a bag of tuples, and finite sets of pending processes with associated attributes. To model template lists we need an auxiliary type definition for optional statements:

$$\mathit{OptStmt} ::= \mathit{NoStmt} \mid \mathit{IsStmt}\langle\!\langle \mathit{Statement} \rangle\!\rangle$$

We shall also use a notion for execution of optional statements based on $\mathit{Execute}$:

$$
\begin{array}{|l}
\mathit{OptExecute}\,_ : \mathbb{P}\ \mathit{OptStmt} \\
\hline
\mathbf{let}\ \mathit{StatementOf} == \mathit{IsStmt}^{-1} \bullet \\
\forall\, os : \mathit{OptStmt} \mid os \in \operatorname{ran} \mathit{IsStmt} \bullet \\
\quad \mathit{OptExecute}\ os \Leftrightarrow \mathit{Execute}\,(\mathit{StatementOf}\ os)
\end{array}
$$

Pending processes are associated with templates and optional statements as attributes for blocking `fetch` and `meet` operations:

$$
\begin{array}{|l}
\textit{Pending} \\
\hline
proc : \mathit{Process} \\
temp : \mathit{Template} \\
os : \mathit{OptStmt} \\
\hline
\end{array}
$$

A tuple space consists of an identity, its specified limit, a bag of tuples, and finite sets of pending processes with associated attributes:

$$
\begin{array}{|l}
\textit{TupleSpace} \\
\hline
\mathit{Id} : \mathit{Value} \\
\mathit{Limit} : \mathit{Value} \\
\mathit{Tuples} : \operatorname{bag} \mathit{APTuple} \\
\mathit{PendFetch} : \mathbb{F}\ \mathit{Pending} \\
\mathit{PendMeet} : \mathbb{F}\ \mathit{Pending} \\
\mathit{PendFull} : \mathit{Process} \mathrel{+\!\!\!\!\!\twoheadrightarrow} \mathit{APTuple} \\
\hline
(\mathit{Type}\ \mathit{Id} = atom) \land (\mathit{Id} \notin \operatorname{dom} \mathit{ValuesOfType}) \\
(\mathit{Type}\ \mathit{Limit} = integer) \lor (\mathit{Type}\ \mathit{Limit} = om) \\
\mathsf{disjoint}\ \langle \{ pf : \mathit{PendFetch} \bullet pf.proc \}, \{ pm : \mathit{PendMeet} \bullet pm.proc \}, \operatorname{dom} \mathit{PendFull} \rangle \\
\hline
\end{array}
$$

Z: $\operatorname{bag} X$ is the set of bags of elements of X:

$$\operatorname{bag} X == X \mathrel{+\!\!\!\!\!\twoheadrightarrow} \mathbb{N}_1$$

where $X \rightarrowtail\!\!\!\!\!\!+ \mathbb{N}_1$ is the set of partial functions from X to $\mathbb{N}_1$. $\mathbb{F}\, S$ is the set of finite subsets of S. $X \twoheadrightarrow\!\!\!\!\!\!+\!\!+ Y$ is the set of finite partial functions from X to Y. An indexed family of sets is disjoint if and only if each pair of sets $S(i)$ and $S(j)$ for $i \neq j$ have empty intersection:

$$\mathsf{disjoint}\ \langle A, B\rangle \Leftrightarrow A \cap B = \{\}$$

Z

Tuple-space identities are atoms (not including the predefined type-atoms). The limit for the number of simultaneously deposited tuples in tuple space has to be an integer or the undefined value. A negative limit is equivalent to 0 (no tuples may be deposited into such a tuple space). The undefined value indicates that no limit has been specified on creation of the tuple space. This limit is a parameter to the function `CreateTS` (section 2.5.1). The main part of a *TupleSpace* it the bag of *APTuples*. The processes pending for `fetch` operations are collected in the finite sets *PendFetch* of the corresponding *TupleSpaces*. The processes pending for `meet` operations are collected in the finite sets *PendMeet* of the corresponding *TupleSpaces*. The processes pending for `deposit` operations on full *TupleSpaces* are collected in the finite sets *PendFull* of the corresponding *TupleSpaces*. A process is at most pending for one of these sets.

Note that we use the terms pending and blocked as synonyms. A process is *pending* if it has executed a `fetch` or `meet` operation with no matching tuple in tuple space and if no `else` statements were specified (blocking matching). A process is also *pending* for a `deposit` operation on a full tuple space provided that `blockiffull` has been specified and the tuple space is full (see below). Such processes may be reactivated by appropriate events, i.e. fetching of tuples.

2.4 Programs and processes

In this section we will define our program state and the creation and termination of programs and processes. We view the state of a program as the state of a finite set of tuple spaces, and a finite set of active processes:

$$
\begin{array}{l}
\textit{Program} \\
\hline
\textit{TSs} : \mathbb{F}\ \textit{TupleSpace} \\
\textit{ActiveProcs} : \mathbb{F}\ \textit{Process} \\
\hline
\forall\, ts1, ts2 : \textit{TSs} \bullet \\
\quad ts1 \neq ts2 \Rightarrow ts1.\textit{Id} \neq ts2.\textit{Id} \\
\forall\, ts1, ts2 : \textit{TSs} \bullet \\
\quad \forall\, tup1, tup2 : \mathrm{dom}(ts1.\textit{Tuples} \uplus ts2.\textit{Tuples});\ i1, i2 : \mathbb{N} \mid \\
\qquad\qquad tup1(i1) \in \mathrm{ran}\ \textit{TupleProcess} \land tup2(i2) \in \mathrm{ran}\ \textit{TupleProcess} \bullet \\
\qquad (i1 \in \mathrm{dom}\, tup1 \land i2 \in \mathrm{dom}\, tup2 \land tup1(i1) = tup2(i2)) \Rightarrow \\
\qquad\quad tup1 = tup2 \land i1 = i2 \land \\
\qquad\quad 1 = (ts1.\textit{Tuples} \uplus ts2.\textit{Tuples}) \sharp\ tup1
\end{array}
$$

Z $\uplus$ denotes the union of bags where the number of times any object appears in the result is the sum of the number of times it appears in the operands. The number of times x appears in the bag B is $B \sharp x$. Z

For a specification of the entire ProSet language, additional components would be necessary to specify the program state.

The first property asserts the uniqueness of tuple-space identities and the second property asserts the uniqueness of processes inside of active tuples. Only processes within *ActiveProcs* are active and executing. Processes, which are suspended for templates or full tuple spaces, are temporally moved to the sets of pending processes of the corresponding tuple spaces. They are reactivated by

returning them to *ActiveProcs*. If a process is removed from *ActiveProcs* and moved to a set of pending processes, it is suspended. If a process is removed from *ActiveProcs* and *not* moved to a set of pending processes, it is terminated (see below). We represent this formally by the global variables *ActuallyActiveProcesses*, *ActuallyPendingProcesses*, and *ActuallyExistingProcesses*:

$$
\begin{array}{|l}
ActuallyActiveProcesses : \mathbb{F}\ Process \\
ActuallyPendingProcesses : \mathbb{F}\ Process \\
ActuallyExistingProcesses : \mathbb{F}\ Process \\
\hline
\forall\ Program;\ p : Process \bullet \\
\quad ActuallyActiveProcesses = ActiveProcs\ \land \\
\quad (p \in ActuallyPendingProcesses \Leftrightarrow \\
\qquad (\exists\ ts : TSs \bullet \\
\qquad\quad (\exists\ pfm : (ts.PendFetch \cup ts.PendMeet) \bullet p = pfm.proc) \lor \\
\qquad\quad (\exists\ pf : ts.PendFull \bullet p = first\ pf))) \\
\langle ActuallyActiveProcesses, ActuallyPendingProcesses \rangle\ \mathsf{partition}\ ActuallyExistingProcesses
\end{array}
$$

Z If p is an ordered pair then $p = (first\ p, second\ p)$ holds. Z

We now start with the specification of operations on the program state. A main process is started for the main program on program initialization:

$$
\begin{array}{|l}
\multicolumn{1}{l}{InitProgram} \\
\hline
\Delta Program \\
\hline
\#ActiveProcs' = 1 \\
TSs' = \{\} \\
\hline
\end{array}
$$

The initial state is primed, because we can regard the initialization of a system as a peculiar kind of operation that creates a state out of nothing; there is no before state, simply an after state, with its variables decorated [Diller, 1990].

We will specify the connection between the computation and the coordination part of ProSet by means of input and output variables of the individual operations. These input and output variables are decorated with ? and !, respectively.

Whenever the process creator || is applied in a ProSet program, this process shall be added to the set of active processes:

$$
\begin{array}{|l}
\multicolumn{1}{l}{ProcessCreation} \\
\hline
\Delta Program \\
NewProcess? : Process \\
\hline
ActiveProcs' = ActiveProcs \cup \{NewProcess?\} \\
TSs' = TSs \\
\hline
\end{array}
$$

Such a process is implicitly added to *ActuallyActiveProcesses* and to *ActuallyExistingProcesses*. These sets and *ActuallyPendingProcesses* are not really necessary for our specification. We introduced them to have a formal specification for actually active, pending, and existing processes. Analogously, when a process that is not the process for the main program terminates, it has to be removed from the set of active processes and from the sets of pending processes.

We shall need an operation for schema anti-restriction for pending processes:

$$
\begin{array}{|l}
_ \boxminus _ : \mathbb{F}\ Process \times \mathbb{F}\ Pending \longrightarrow \mathbb{F}\ Pending \\
\hline
\forall\ procs : \mathbb{F}\ Process;\ pends : \mathbb{F}\ Pending \bullet \\
\quad procs \boxminus pends = \\
\qquad \{\ pr : Process;\ pe : Pending \mid pr \notin procs \land pr = pe.proc \land pe \in pends \bullet pe\ \}
\end{array}
$$

The schema anti-restriction $PR \boxbar PE$ yields all the members of PE, where the *proc* component is not a member of PR. We call $\boxbar$ schema anti-restriction, because of its similarity to the domain anti-restriction $\ndres$ of Z (see below).

If a process, which has to be removed, is part of an active tuple, it has to be replaced by the corresponding return value:

ProcessTermination

$$
\begin{array}{l}
\Delta Program \\
ToKill? : Process \\
\hline
TSs' = \{\, ts : TSs;\ ts' : TupleSpace \mid ts.Id = ts'.Id \land \\
\qquad ts'.Limit = ts.Limit \land \\
\qquad ts'.PendFetch = \{ToKill?\} \boxbar ts.PendFetch \land \\
\qquad ts'.PendMeet = \{ToKill?\} \boxbar ts.PendMeet \land \\
\qquad ts'.PendFull = \{ToKill?\} \ndres ts.PendFull \land \\
\qquad (\forall\, tup : \mathrm{dom}\, ts.Tuples;\ newtup : APTuple \mid \#tup = \#newtup \land \\
\qquad\qquad\qquad (\exists\, tupc : \mathrm{ran}\, tup \bullet tupc \in \mathrm{ran}\, TupleProcess) \bullet \\
\qquad\quad (\forall\, i : \mathrm{dom}\, tup \bullet \\
\qquad\qquad newtup(i) = \mathbf{if}\ tup(i) = TupleProcess\ ToKill? \\
\qquad\qquad\qquad \mathbf{then}\ TupleValue\ (ProcRetVal\ ToKill?) \\
\qquad\qquad\qquad \mathbf{else}\ tup(i)) \land \\
\qquad\quad ts'.Tuples = (ts.Tuples \uminus \llbracket tup \rrbracket) \uplus \llbracket newtup \rrbracket) \\
\qquad\qquad \bullet ts' \,\} \\
ActiveProcs' = ActiveProcs \setminus \{ToKill?\}
\end{array}
$$

Z The domain anti-restriction $S \ndres R$ of a relation R to a set S relates x to y only if R relates x to y and x is not a member of S. We write $\llbracket a_1, \ldots, a_n \rrbracket$ for the bag $\{a_1 \mapsto k_1, \ldots, a_n \mapsto k_n\}$ where the elements a_i appear k_i times. For instance: $\llbracket 1, 1 \rrbracket = \{1 \mapsto 2\}$. The empty bag is $\llbracket\, \rrbracket$. $B \uminus C$ is the bag difference of B and C: the number of times any object appears in it is the number of times it appears in B minus the number of times it appears in C, or zero if that would be negative. **Z**

As you can see, such a terminated process is not only removed from *ActiveProcs*. It is also necessary to remove it from the sets of pending processes in the tuple spaces, and to resolve the future within an active tuple, provided that this process has been spawned as a component of this active tuple. Future resolution is modeled within the above conditional expression. See section D.1.2 for an informal specification for resolving of futures in ProSet. The resolving and touching of futures is only specified for processes within active tuples in tuple space, and not for processes spawned outside of tuple space. This limitation is due to the fact that this paper is not a specification of the entire language.

Every time a process is removed from the set of active processes and not moved to the sets of pending processes, this process will be terminated. Whenever the process for the main program terminates, the entire program terminates (see appendix D):

ProgramTermination

$$
\begin{array}{l}
\Delta Program \\
\hline
ActiveProcs' = \{\} \land TSs' = \{\}
\end{array}
$$

2.5 Multiple tuple spaces

In this section we define the library functions to handle multiple tuple spaces as informally specified in section D.3.

2.5.1 CreateTS

At first we define the auxiliary function *IDsOF*, which yields the finite set of tuple-space identities from a finite set of tuple spaces:

$$IDsOF : \mathbb{F}\ TupleSpace \longrightarrow \mathbb{F}\ Value$$

$$\forall\ tss : \mathbb{F}\ TupleSpace \bullet$$
$$\quad IDsOF\ tss = \{\ id : Value \mid (\exists\ ts : tss \bullet ts.Id = id)\ \}$$

The library function `CreateTS` creates a new tuple space and returns the corresponding tuple-space identity, provided that the given limit is an integer or the undefined value:

CreateTSok

$$\Delta Program$$
$$InLimit? : Value$$
$$Return! : Value$$

$$(Type\ InLimit? = integer) \lor (Type\ InLimit? = om)$$

$$\exists_1\ a : Value;\ ts : TupleSpace \mid$$
$$\quad (Type\ a = atom) \land (a \notin IDsOF\ TSs) \land (a \notin \mathrm{dom}\ ValuesOfType) \land$$
$$\quad ts.Id = a \land$$
$$\quad ts.Limit = InLimit? \land$$
$$\quad ts.Tuples = [\![\,]\!] \land$$
$$\quad ts.PendFetch = \{\} \land$$
$$\quad ts.PendMeet = \{\} \land$$
$$\quad ts.PendFull = \{\} \bullet$$
$$TSs' = TSs \cup \{ts\} \land$$
$$ActiveProcs' = ActiveProcs \land$$
$$Return! = a$$

Z The predicate $\exists_1\ S \bullet P$ is true if there is exactly one way of giving values to the variables introduced by S so that both the property S and the predicate P are true. Z

A new, empty tuple space is created this way. The tuple-space identity (an atom) will be created via a call to the standard library function **newat**, which returns a new, unique atom [Doberkat *et al.*, 1992]. For our specification it is sufficient to specify that this tuple-space identity is unique with respect to our program state. The new tuple-space identity will be returned.

The ProSet-statement '`escape type_mismatch();`' raises the exception `type_mismatch`, which should not be resumed:

'`escape type_mismatch();`' : *Statement*

Z Generic definitions uniquely determine the value of the introduced global constant. We use generic definitions to introduce exceptions, because there is nothing more to be said about them. Z

The given limit for the number of simultaneously deposited tuples in tuple space has to be an integer or the undefined value:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CreateTSTypeMismatch \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Xi Program \\
InLimit? : Value \\
Exception! : Statement \\
\hline
\neg\ ((Type\ InLimit? = integer) \lor (Type\ InLimit? = om)) \\
Exception! = \text{`}\texttt{escape type_mismatch();}\text{'} \\
\hline
\end{array}
$$

Exception handling is specified for operations, which may change the state, in separate schemas with $\Xi Program$ instead of $\Delta Program$ to emphasize that the state does not change when such errors occur. The disjunction of *CreateTSok* and *CreateTSTypeMismatch* then constitutes *CreateTS*:

$$CreateTS \mathrel{\widehat{=}} CreateTSok \lor CreateTSTypeMismatch$$

This definition introduces a new schema called *CreateTS*, obtained by combining the two schemas on the right-hand side.

> Z The operation *CreateTS* could be specified directly by writing a single schema which combines the predicate parts of the two schemas *CreateTSok* and *CreateTSTypeMismatch*. The effect of the schema $\lor$ operator is to make a schema in which the predicate part is the result of joining the predicate parts of its arguments with the logical connective $\lor$. Similarly, the effect of the schema $\land$ operator is to take the conjunction of the two predicate parts. Any common variables of the two schemas are merged. We sketch an alternative specification of *CreateTS*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CreateTS \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta Program \\
InLimit? : Value \\
Return! : Value \\
Exception! : Statement \\
\hline
((Type\ InLimit? = integer) \lor (Type\ InLimit? = om) \land \\
\qquad \exists_1\ a : Value; ts : TupleSpace \mid \\
\qquad\qquad \ldots) \\
\qquad\qquad\qquad \lor \\
(\neg\ ((Type\ InLimit? = integer) \lor (Type\ InLimit? = om)) \land \\
\qquad Exception! = \text{`}\texttt{escape type_mismatch();}\text{'} \land \\
\qquad TSs' = TSs \land \\
\qquad ActiveProcs' = ActiveProcs) \\
\hline
\end{array}
$$

> In order to write *CreateTS* as a single schema, it has been necessary to write out explicitly that the state does not change when an exception is raised. Z

We do not specify the return values when exceptions are raised. Those return values depend on the actions taken by the associated exception handler.

2.5.2 ExistsTS

The library function `ExistsTS` checks if a given atom is a valid tuple-space identity:

$$
\begin{array}{|l}
\hline
\textit{ExistsTS} \\
\Xi Program \\
InTS? : Value \\
Return! : Value \\
Exception! : Statement \\
\hline
(Type\ InTS? = atom \land Return! = \mathbf{if}\ InTS? \in IDsOF\ TSs \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ TRUE \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ FALSE) \\
\quad \lor \\
(Type\ InTS? \neq atom \land Exception! = \text{`}\texttt{escape type_mismatch();}\text{'}) \\
\hline
\end{array}
$$

ExistsTS does not modify the program state. It returns a Boolean *Value* indicating the existence of the given tuple-space identity, which has to be an atom.

2.5.3 ClearTS

The library function `ClearTS` removes all active and passive tuples from a specified tuple space:

$$
\begin{array}{|l}
\hline
\textit{ClearTSok} \\
\Delta Program \\
InTS? : Value \\
Return! : Value \\
\hline
InTS? \in IDsOF\ TSs \\
\mathbf{let}\ ClearProcs == \{\, ts : TSs;\ p : ActiveProcs \mid ts.Id = InTS? \land \\
\qquad\qquad (\exists\, tup : \mathrm{dom}\, ts.Tuples \bullet \exists\, tupc : \mathrm{ran}\, tup \bullet tupc = TupleProcess\ p) \bullet p \,\} \bullet \\
TSs' = \{\, ts : TSs;\ ts' : TupleSpace \mid ts.Id = ts'.Id \land \\
\qquad ts'.Limit = ts.Limit \land \\
\qquad ts'.Tuples = \mathbf{if}\ ts'.Id = InTS? \\
\qquad\qquad\qquad \mathbf{then}\ [\![\,]\!] \\
\qquad\qquad\qquad \mathbf{else}\ ts.Tuples \land \\
\qquad ts'.PendFetch = ClearProcs \mathbin{⩁} ts.PendFetch \land \\
\qquad ts'.PendMeet = ClearProcs \mathbin{⩁} ts.PendMeet \land \\
\qquad ts'.PendFull = ClearProcs \mathbin{⩤} ts.PendFull \bullet ts' \,\} \land \\
ActiveProcs' = ActiveProcs \setminus ClearProcs \\
Return! = om \\
\hline
\end{array}
$$

The processes within removed active tuples (the set *ClearProcs*) are also removed from *ActiveProcs* and the sets of pending processes, and thus terminated. Note that the processes, which are not in *ClearProcs*, are **not** removed from the sets of pending processes of *InTS?*. Only the bag *Tuples* is cleared in *InTS?*. The return value is the undefined value in any case.

The exception `ts_invalid_id` will be raised when invalid tuple-space identities are given:

$$
\begin{array}{|l}
\hline
\text{`}\texttt{escape ts_invalid_id();}\text{'} : Statement \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ClearTSinvalid} \\
\Xi \mathit{Program} \\
\mathit{InTS?} : \mathit{Value} \\
\mathit{Exception!} : \mathit{Statement} \\
\hline
\mathit{InTS?} \notin \mathit{IDsOF}\ \mathit{TSs} \\
\mathit{Exception!} = \mathbf{if}\ \mathit{Type}\ \mathit{InTS?} = \mathit{atom} \\
\qquad\qquad \mathbf{then}\ \text{`}\texttt{escape ts_invalid_id();}\text{'} \\
\qquad\qquad \mathbf{else}\ \text{`}\texttt{escape type_mismatch();}\text{'} \\
\hline
\end{array}
$$

Note that the exception `type_mismatch` and not `ts_invalid_id` will be raised if the actual parameter for `ClearTS` is *not* an atom.

The disjunction of *ClearTSok* and *ClearTSinvalid* then constitutes *ClearTS*:

$$\mathit{ClearTS} \mathrel{\widehat{=}} \mathit{ClearTSok} \lor \mathit{ClearTSinvalid}$$

2.5.4 RemoveTS

The library function `RemoveTS` calls `ClearTS` and removes the given tuple space from the set of tuple spaces. First we specify the auxiliary operation *RemoveTSfromState*:

$$
\begin{array}{|l}
\hline
\textit{RemoveTSfromState} \\
\Delta \mathit{Program} \\
\mathit{InTS?} : \mathit{Value} \\
\hline
\mathit{TSs}' = \mathit{TSs} \setminus \{\, \mathit{ts} : \mathit{TSs} \mid \mathit{ts}.\mathit{Id} = \mathit{InTS?} \,\} \\
\mathit{ActiveProcs}' = \mathit{ActiveProcs} \\
\hline
\end{array}
$$

Therefore, the entire tuple space is removed from the program state. This may terminate pending processes, which are only blocked on *InTS?*, provided that they are not pending on other tuple spaces.

RemoveTS is the sequential composition of *ClearTS* and *RemoveTSfromState*:

$$\mathit{RemoveTS} \mathrel{\widehat{=}} \mathit{ClearTS} \mathbin{⨾} \mathit{RemoveTSfromState}$$

[Z] If $Op1$ and $Op2$ are schemas describing two operations, then $Op1 ⨾ Op2$ is a schema which describes their sequential composition. The components of $Op1 ⨾ Op2$ are the undecorated components of $Op1$ and the primed components of $Op2$, together with their merged inputs and outputs. Conversely, the components of the piping $Op1 \gg Op2$ are the inputs of $Op1$ and the outputs of $Op2$, together with their merged decorated and undecorated components (the outputs of $Op1$ have to match the inputs of $Op2$). [Z]

2.6 Tuple-space operations

A concise overview of the abstract grammar for the tuple-space operations is displayed in Fig. 1 using BNF (Backus Naur From). Conversely, the informal semantics in appendix D is presented together with syntax diagrams, which are spread over the text. For a concise overview, we regard BNF as more appropriate. This grammar is not part of the present Z specification. Optional parts are enclosed in *[* and *]*. Terminal symbols are displayed in `typewriter` font. Note that the terminal symbol `|` is different from |, which denotes alternatives in the grammar.

Section 2.6.1 provides some preliminary definitions for the *Deposit* operation, which is defined in section 2.6.2. Sections 2.6.3 and 2.6.4 will define the *Fetch* and *Meet* operations, respectively.

```
Statement    ::= deposit DepositArgs end deposit ;
               | fetch FetchList [ else StmtList ] end fetch ;
               | meet MeetList [ else StmtList ] end meet ;

DepositArgs  ::= DepositList
               | blockiffull Expr at Expr

DepositList  ::= ExprList at Expr [ also DepositList ]

FetchList    ::= FetchTemp at Expr [ xor FetchList ]

FetchTemp    ::= FetchTemp [ xor FetchTemp ]
               | ( [ FetchComp ] ) [ => StmtList ]

FetchComp    ::= FetchComp [ , FetchComp ]
               | Expr
               | ? [ LValue ] [ | Expr ]

MeetList     ::= MeetTemp at Expr [ xor MeetList ]

MeetTemp     ::= MeetTemp [ xor MeetTemp ]
               | ( [ MeetComp ] ) [ => StmtList ]

MeetComp     ::= MeetComp [ , MeetComp ]
               | Expr
               | ? [ LValue ] [ | Expr ] [ into Expr ]

ExprList     ::= Expr [ , ExprList ]

StmtList     ::= Statement [ StmtList ]
```

Figure 1: The abstract grammar for the tuple-space operations.

2.6.1 Some preliminary definitions

To begin with a description of the tuple-space operations we need some preliminary definitions. A `deposit` operation such as

```
deposit t1, t2 at TS1
   also t3 at TS2
end deposit;
```

produces

$$\langle(\langle \mathtt{t1}, \mathtt{t2}\rangle, \mathtt{TS1}), (\langle \mathtt{t3}\rangle, \mathtt{TS2})\rangle$$

as input for our *Deposit* specification (see below) with type:

$TupList == \mathrm{seq}_1((\mathrm{seq}_1\ APTuple) \times Value)$

Ⓩ $\mathrm{seq}_1\ X = \mathrm{seq}\ X \setminus \{\langle\rangle\}$ Ⓩ

We shall use the predicate *HasIntos* for the pending processes of tuple spaces, which checks if there are `into` expressions associated:

$HasIntos\,_ : \mathbb{P}\ Template$

$\mathbf{let}\ FormalOf == TempFormal^{-1} \bullet$
$\forall\ pl : Pending \bullet$
$\quad HasIntos\ pl.temp \Leftrightarrow$
$\quad\quad (\exists\ tempc : \mathrm{ran}\ pl.temp.List \mid tempc \in \mathrm{ran}\ TempFormal \bullet$
$\quad\quad\quad (FormalOf\ tempc).Into \neq NoInto)$

We define two auxiliary functions for controlling the limits of a tuple space. The function AllTuples extracts from a given *TupList* all sequences of *APTuples* for a given tuple-space identity (*Value*). The found sequences for the given tuple-space identity are concatenated to yield a result sequence of *APTuples*:

$_\ \mathsf{AllTuples}\ _ : (Value \times TupList) \longrightarrow \mathrm{seq}\ APTuple$

$\forall\ id : Value;\ tupl : TupList;\ c : (\mathrm{seq}_1\ APTuple) \times Value \bullet$
$\quad id\ \mathsf{AllTuples}\ \langle\rangle = \langle\rangle \land$
$\quad id\ \mathsf{AllTuples}\ (\langle c\rangle \frown tupl) = \mathbf{if}\ id \neq second\ c$
$\quad\quad\quad \mathbf{then}\ \langle\rangle \frown (id\ \mathsf{AllTuples}\ (tail\ tupl))$
$\quad\quad\quad \mathbf{else}\ (first\ c) \frown (id\ \mathsf{AllTuples}\ (tail\ tupl))$

Ⓩ $\frown$ denotes the concatenation of sequences. The sequences $tail\ S$ and $front\ S$ contain all the elements of the non-empty sequence S, except for the first and except for the last element, respectively. *head* yields the first component of a non-empty sequence. *last* yields the last component of a non-empty sequence. Ⓩ

We need an auxiliary function that yields the number of elements in a bag to control the limits of tuple spaces:

$[X]$

$BagSum : \mathrm{bag}\ X \longrightarrow \mathbb{N}$

$\forall\ b : \mathrm{bag}\ X;\ x : X \bullet$
$\quad BagSum\ [\![\,]\!] = 0 \land$
$\quad BagSum\ ([\![x]\!] \uplus b) = 1 + BagSum\ b$

A function that yields the $\mathbb{Z}$-integer value from a *Value* with type *integer*:

$$
\begin{array}{|l}
IntValueOf : Value \longrightarrow \mathbb{Z} \\
\hline
\forall\, i : Value \mid Type\ i = integer \bullet \\
\quad \exists\, z : \mathbb{Z} \bullet IntValueOf\ i = z
\end{array}
$$

Condition for full tuple spaces:

$$
\begin{array}{|l}
\multicolumn{1}{l}{TSisFull} \\
\hline
\Xi Program \\
InTupList? : TupList \\
\hline
\exists\, pair : \mathrm{ran}\, InTupList?;\ ts : TSs \mid ts.Id = second\ pair \bullet \\
\quad Type\ ts.Limit = integer\ \land \\
\quad IntValueOf\ ts.Limit < (BagSum\ ts.Tuples + \#(ts.Id\ \mathsf{AllTuples}\ InTupList?)) \\
\hline
\end{array}
$$

This schema will later be composed with some schema components of *Deposit*.

2.6.2 Depositing tuples

This section defines the *Deposit* operation incrementally. The informal specification is given in section D.2.1. As a first step we define an operation for adding a single tuple to a specified tuple space of a program:

$$
\begin{array}{l}
_\ \mathsf{AddTuple}\ _ : (Program \times (APTuple \times Value)) \longrightarrow Program \\
\hline
\forall \Delta Program; tup : APTuple; id : Value \mid id \in IDsOF\ TSs \bullet \\
\quad \theta Program' = \theta Program\ \mathsf{AddTuple}\ (tup, id) \Leftrightarrow \\
\quad\quad (\mathbf{let}\ MatchMeets == \{ ts : TSs; pdg : Pending \mid (\exists pm : ts.PendMeet \bullet \\
\quad\quad\quad\quad tup\ \mathsf{Matches}\ pm.temp \land \neg\ HasIntos\ pm.temp \land pm = pdg) \bullet pdg \}; \\
\quad\quad\quad MatchIntos == \{ ts : TSs; pdg : Pending \mid (\exists pm : ts.PendMeet \bullet \\
\quad\quad\quad\quad tup\ \mathsf{Matches}\ pm.temp \land HasIntos\ pm.temp \land pm = pdg) \bullet pdg \}; \\
\quad\quad\quad MatchFetchs == \{ ts : TSs; pdg : Pending \mid (\exists pf : ts.PendFetch \bullet \\
\quad\quad\quad\quad tup\ \mathsf{Matches}\ pf.temp \land pf = pdg) \bullet pdg \}; \\
\quad\quad\quad NewProcs == \{ p : Process \mid (\exists tupc : \mathrm{ran}\ tup \bullet tupc = TupleProcess\ p) \} \bullet \\
\quad\quad (\forall mm : MatchMeets \bullet \\
\quad\quad\quad mm.temp\ \mathsf{FormalAssign}\ tup \land \\
\quad\quad\quad OptExecute\ mm.os) \land \\
\quad\quad ((MatchIntos \cup MatchFetchs = \{\}) \Rightarrow \\
\quad\quad\quad (TSs' = \{ ts : TSs; ts' : TupleSpace \mid ts'.Id = ts.Id \land \\
\quad\quad\quad\quad\quad ts'.Limit = ts.Limit \land \\
\quad\quad\quad\quad\quad ts'.Tuples = \mathbf{if}\ ts'.Id = id \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{then}\ ts.Tuples \uplus [\![tup]\!] \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{else}\ ts.Tuples \land \\
\quad\quad\quad\quad\quad ts'.PendMeet = \{ mm : MatchMeets \bullet mm.proc \} ⩤ ts.PendMeet \land \\
\quad\quad\quad\quad\quad ts'.PendFetch = ts.PendFetch \land \\
\quad\quad\quad\quad\quad ts'.PendFull = ts.PendFull \bullet ts' \} \land \\
\quad\quad\quad ActiveProcs' = \\
\quad\quad\quad\quad\quad ActiveProcs \cup NewProcs \cup \{ mm : MatchMeets \bullet mm.proc \})) \land \\
\quad\quad ((MatchIntos \cup MatchFetchs \neq \{\}) \Rightarrow \\
\quad\quad (\exists mif : (MatchIntos \cup MatchFetchs) \bullet \\
\quad\quad\quad mif.temp\ \mathsf{FormalAssign}\ tup \land \\
\quad\quad\quad OptExecute\ mif.os \land \\
\quad\quad\quad (mif \in MatchFetchs \Rightarrow \\
\quad\quad\quad\quad (TSs' = \{ ts : TSs; ts' : TupleSpace \mid ts'.Id = ts.Id \land \\
\quad\quad\quad\quad\quad\quad ts'.Limit = ts.Limit \land \\
\quad\quad\quad\quad\quad\quad ts'.Tuples = ts.Tuples \land \\
\quad\quad\quad\quad\quad\quad ts'.PendMeet = \\
\quad\quad\quad\quad\quad\quad\quad \{ mm : MatchMeets \bullet mm.proc \} ⩤ ts.PendMeet \land \\
\quad\quad\quad\quad\quad\quad ts'.PendFetch = \{mif.proc\} ⩤ ts.PendFetch \land \\
\quad\quad\quad\quad\quad\quad ts'.PendFull = ts.PendFull \bullet ts' \} \land \\
\quad\quad\quad\quad ActiveProcs' = \\
\quad\quad\quad\quad\quad ActiveProcs \cup \{ mm : MatchMeets \bullet mm.proc \} \cup \{mif.proc\})) \land \\
\quad\quad\quad (mif \in MatchIntos \Rightarrow \\
\quad\quad\quad (\exists_1 Program'' \bullet \\
\quad\quad\quad\quad TSs'' = \{ ts : TSs; ts'' : TupleSpace \mid ts''.Id = ts.Id \land \\
\quad\quad\quad\quad\quad\quad ts''.Limit = ts.Limit \land \\
\quad\quad\quad\quad\quad\quad ts''.Tuples = ts.Tuples \land \\
\quad\quad\quad\quad\quad\quad ts''.PendMeet = (\{ mm : MatchMeets \bullet mm.proc \} \cup \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \{mif.proc\}) ⩤ ts.PendMeet \land \\
\quad\quad\quad\quad\quad\quad ts''.PendFetch = ts.PendFetch \land \\
\quad\quad\quad\quad\quad\quad ts''.PendFull = ts.PendFull \bullet ts'' \} \land \\
\quad\quad\quad\quad ActiveProcs'' = \\
\quad\quad\quad\quad\quad ActiveProcs \cup \{ mm : MatchMeets \bullet mm.proc \} \cup \{mif.proc\} \land \\
\quad\quad\quad\quad \theta Program' = \theta Program''\ \mathsf{AddTuple}\ (mif.temp\ \mathsf{EvalIntos}\ tup, id))))))
\end{array}
$$

Z An identifier or schema may have a sequence of decorations in Z. In AddTuple we use the components of $Program''$ ($ActiveProcs''$ and TSs'') as auxiliary store for the recursion. Z

Since this is a quite longish formula we give an informal outline of the predicate on the right-hand side of $\Leftrightarrow$ in the previous schema. The local set *MatchMeets* contains all the pending processes which are pending for `meet` operations without `into` expressions, where the associated templates match the tuple to be added. *MatchIntos* contains all the pending processes which are pending for `meet` operations with `into` expressions, where the associated templates match the tuple to be added. *MatchFetchs* contains all the pending processes which are pending for `fetch` operations, where the associated templates match the tuple to be added. *NewProcs* contains the processes which are active within the tuple to be added. *NewProcs* is empty if the tuple to be added matches any template in *TSs*, because only passive tuples can match. An informal outline of the **let** predicate follows:

(for all `meet`s without `into`s that match:
  (assign the tuple fields to the *l*-values in the corresponding formals (optional)) $\wedge$
  (execute the associated statements (optional))
    $\wedge$
(if not exists a `meet` with `into`s or a `fetch`, which matches:
  add the tuple)) (1)
    $\wedge$
(if exists a `meet` with `into`s or a `fetch`, which matches:
  (assign the tuple fields to the *l*-values in the corresponding formals (optional)) $\wedge$
  (execute the associated statements (optional))
  (if a `fetch` was selected:
    satisfy the selected `fetch`) (2)
  (if a `meet` with `into`s was selected:
    satisfy the selected `meet`)) (3)

We expand the three numbered operation parts of this schema. Adding a tuple for which no pending `meet` with `into`s and no pending `fetch` exists (1):

(add the tuple to the the bag-component *Tuples*) $\wedge$
(remove the processes which were pending for `meet`s without `into`s from the `meet`-lists of pending processes) $\wedge$
(add the contained active processes to *ActiveProcs*) $\wedge$
(reactivate the processes which were pending for `meet`s without `into`s)

Satisfaction of a `fetch`, which matches implies the following actions (2):

(remove the processes which were pending for `meet`s without `into`s from the `meet`-lists of pending processes) $\wedge$
(remove the associated process from the `fetch`-lists of pending processes) $\wedge$
(reactivate the processes which were pending for `meet`s without `into`s) $\wedge$
(reactivate the associated process)

If we satisfy a matching, pending `fetch`, we do not add the tuple. Satisfaction of a `meet` with `into`s, which matches implies the following actions (3):

(remove the processes which were pending for `meet`s without `into`s from the `meet`-lists of pending processes) $\wedge$
(remove the associated process from the `meet`-lists of pending processes) $\wedge$
(reactivate the processes which were pending for `meet`s without `into`s) $\wedge$
(reactivate the associated process) $\wedge$
(add the changed tuple recursively with AddTuple)

For a matching `meet` template with `into`s the changed tuple and not the original tuple has to be added. This may satisfy other pending templates, and therefore AddTuple may call itself recursively an arbitrary but finite number of times.

Successful depositing of a list of tuples then becomes straightforward:

DepositOK

$\Delta Program$
$InTupList? : TupList$

$\forall\, pair : \operatorname{ran} InTupList? \bullet second\ pair \in IDsOF\ TSs$

$\forall\, pair : \operatorname{ran} InTupList? \bullet \forall\, tup : \operatorname{ran}(first\ pair) \bullet$
$\quad \theta Program' = \theta Program\ \mathsf{AddTuple}\ (tup, second\ pair)$

The exception `ts_invalid_id` will be raised when invalid tuple-space identities are given:

DepositInvalid

$\Xi Program$
$InTupList? : TupList$
$Exception! : Statement$

$\neg\, (\forall\, pair : \operatorname{ran} InTupList? \bullet second\ pair \in IDsOF\ TSs)$

$Exception! =$ '`escape ts_invalid_id();`'

It is not specified that the exception `type_mismatch` will be raised if the tuple-operands are not tuples. Since *Deposit* deposits passive and active tuples in tuple space, the input-tuples cannot be first-class ProSet-values: we need objects of type *APTuple*. We only indicate exception handling for full tuple spaces as described in appendix D. Some auxiliary definitions:

$BlockMode ::= BlockIfFull \mid DoNotBlock$

'`signal ts_is_full();`' $: Statement$

See section D.2.1 for an informal description of the `signal` statement. The exception `ts_is_full` will then be raised when the requested tuple space is full and `blockiffull` has `not` been specified (*DoNotBlock*):

FullTSException

$\Xi Program$
$InTupList? : TupList$
$Blocking? : BlockMode$
$Exception! : Statement$

$\forall\, pair : \operatorname{ran} InTupList? \bullet second\ pair \in IDsOF\ TSs$

$Blocking? = DoNotBlock$

$Exception! =$ '`signal ts_is_full();`'

The exception `ts_is_full` will only be raised when `blockiffull` has `not` been specified. Blocking on full tuple spaces (provided that `blockiffull` has been specified):

$$
\begin{array}{|l}
\multicolumn{1}{l}{__ \mathit{FullTSBlock} ____________________} \\
\Delta \mathit{Program} \\
\mathit{InTupList?} : \mathit{TupList} \\
\mathit{InProc?} : \mathit{Process} \\
\mathit{Blocking?} : \mathit{BlockMode} \\
\hline
\forall \mathit{pair} : \mathrm{ran}\ \mathit{InTupList?} \bullet \mathit{second\ pair} \in \mathit{IDsOF\ TSs} \\
\mathit{Blocking?} = \mathit{BlockIfFull} \\
\mathit{TSs}' = \{\, \mathit{ts} : \mathit{TSs};\ \mathit{ts}' : \mathit{TupleSpace};\ \mathit{pair} : \mathrm{ran}\ \mathit{InTupList?} \mid \mathit{ts}'.\mathit{Id} = \mathit{ts}.\mathit{Id} \wedge \\
\qquad \mathit{ts}'.\mathit{Limit} = \mathit{ts}.\mathit{Limit} \wedge \\
\qquad \mathit{ts}'.\mathit{Tuples} = \mathit{ts}.\mathit{Tuples} \wedge \\
\qquad \mathit{ts}'.\mathit{PendFetch} = \mathit{ts}.\mathit{PendFetch} \wedge \\
\qquad \mathit{ts}'.\mathit{PendMeet} = \mathit{ts}.\mathit{PendMeet} \wedge \\
\qquad \mathit{ts}'.\mathit{PendFull} = \mathbf{if}\ \mathit{ts}'.\mathit{Id} = \mathit{second\ pair} \\
\qquad\qquad\qquad \mathbf{then}\ \mathit{ts}.\mathit{PendFull} \cup \\
\qquad\qquad\qquad\qquad \{\, \mathit{InProc?} \mapsto \mathit{head}(\mathit{first}(\mathit{head\ InTupList?}))\,\} \\
\qquad\qquad\qquad \mathbf{else}\ \mathit{ts}.\mathit{PendFull} \\
\qquad\quad \bullet \mathit{ts}' \,\} \\
\mathit{ActiveProcs}' = \mathit{ActiveProcs} \setminus \{\mathit{InProc?}\} \\
\hline
\end{array}
$$

The condition for full tuple spaces (*TSisFull*) will be added to *FullTSException* and *FullTSBlock* in the schema composition for *Deposit* (see below).

When `blockiffull` has been specified, the tuple list has to contain exactly one tuple:

$$
\begin{array}{l}
\forall \mathit{FullTSBlock} \bullet \\
\qquad 1 = \#\mathit{InTupList?} = \#(\mathit{first}(\mathit{head\ InTupList?}))
\end{array}
$$

Note that `ts_invalid_id` and not `ts_is_full` will be raised if both exceptional conditions — invalid tuple-space identity and full tuple space — hold.

The following schema composition then constitutes *Deposit*:

$$
\begin{array}{ll}
\mathit{Deposit} \mathrel{\widehat{=}} & (\mathit{DepositOK} \wedge \neg\ \mathit{TSisFull}) \\
& \qquad\quad \vee \\
& ((\mathit{FullTSException} \vee \mathit{FullTSBlock}) \wedge \mathit{TSisFull}) \\
& \qquad\quad \vee \\
& \mathit{DepositInvalid}
\end{array}
$$

2.6.3 Fetching tuples

This section defines the *Fetch* operation incrementally. The informal specification is given in section D.2.2. The structure of the input for the *Fetch* operation is similar to the input for the *Deposit* operation, except that tuples are replaced by templates with associated optional statements:

$$
\mathit{TempList} == \mathrm{seq}_1(\mathrm{seq}_1(\mathit{Template} \times \mathit{OptStmt}) \times \mathit{Value})
$$

We shall need some auxiliary projection functions for Z-tuples with four components:

$$
\begin{array}{l}
[A, B, C, D] \\
\hline
GetTS : A \times B \times C \times D \longrightarrow A \\
GetTemp : A \times B \times C \times D \longrightarrow B \\
GetTup : A \times B \times C \times D \longrightarrow C \\
GetOS : A \times B \times C \times D \longrightarrow D \\
\hline
\forall\, a : A;\ b : B;\ c : C;\ d : D \bullet \\
\quad GetTS(a, b, c, d) = a \wedge \\
\quad GetTemp(a, b, c, d) = b \wedge \\
\quad GetTup(a, b, c, d) = c \wedge \\
\quad GetOS(a, b, c, d) = d
\end{array}
$$

Fetching of a matching tuple:

$$
\begin{array}{l}
FetchMatch \\
\hline
\Delta Program \\
InTempList? : TempList \\
InProc? : Process \\
\hline
\forall\, pair : \mathrm{ran}\ InTempList? \bullet second\ pair \in IDsOF\ TSs \\[1ex]
\exists\, pair : \mathrm{ran}\ InTempList?;\ ts : TSs \mid (ts.Id = second\ pair) \bullet \\
\quad \exists\, tos : \mathrm{ran}(first\ pair);\ tup : \mathrm{dom}\ ts.Tuples \bullet tup\ \mathsf{Matches}\ (first\ tos) \\[1ex]
\mathbf{let}\ Matchings == \{\, ts : TSs;\ pair : \mathrm{ran}\ InTempList?;\ TEMP : Template; \\
\quad TUP : APTuple;\ OS : OptStmt \mid (\exists\, tos : \mathrm{ran}(first\ pair);\ tup : \mathrm{dom}\ ts.Tuples \bullet \\
\quad tup\ \mathsf{Matches}\ (first\ tos) \wedge TEMP = first\ tos \wedge TUP = tup \wedge OS = second\ tos) \\
\qquad \bullet (ts, TEMP, TUP, OS) \,\} \bullet \\
(\exists\, SelMatch : Matchings \bullet \\
\quad GetTemp\ SelMatch\ \mathsf{FormalAssign}\ GetTup\ SelMatch \wedge \\
\quad OptExecute\ (GetOS\ SelMatch) \wedge \\
\quad (((GetTS\ SelMatch).PendFull = \{\}) \Rightarrow \\
\qquad (TSs' = \{\, ts : TSs;\ ts' : TupleSpace \mid ts.Id = ts'.Id \wedge \\
\qquad\qquad ts'.Limit = ts.Limit \wedge \\
\qquad\qquad ts'.Tuples = \mathbf{if}\ ts'.Id = (GetTS\ SelMatch).Id \\
\qquad\qquad\qquad\qquad \mathbf{then}\ ts.Tuples \uplus [\![GetTup\ SelMatch]\!] \\
\qquad\qquad\qquad\qquad \mathbf{else}\ ts.Tuples \wedge \\
\qquad\qquad ts'.PendFetch = ts.PendFetch \wedge \\
\qquad\qquad ts'.PendMeet = ts.PendMeet \wedge \\
\qquad\qquad ts'.PendFull = ts.PendFull \bullet ts' \,\} \wedge \\
\qquad ActiveProcs' = ActiveProcs)) \wedge \\
\quad (((GetTS\ SelMatch).PendFull \neq \{\}) \Rightarrow \\
\quad (\exists\, SelBlocked : (GetTS\ SelMatch).PendFull \bullet \\
\qquad (TSs' = \{\, ts : TSs;\ ts' : TupleSpace \mid ts.Id = ts'.Id \wedge \\
\qquad\qquad ts'.Limit = ts.Limit \wedge \\
\qquad\qquad ts'.Tuples = \mathbf{if}\ ts'.Id = (GetTS\ SelMatch).Id \\
\qquad\qquad\qquad\qquad \mathbf{then}\ (ts.Tuples \uplus [\![GetTup\ SelMatch]\!]) \\
\qquad\qquad\qquad\qquad\qquad\qquad \uplus [\![second\ SelBlocked]\!] \\
\qquad\qquad\qquad\qquad \mathbf{else}\ ts.Tuples \wedge \\
\qquad\qquad ts'.PendFetch = ts.PendFetch \wedge \\
\qquad\qquad ts'.PendMeet = ts.PendMeet \wedge \\
\qquad\qquad ts'.PendFull = \mathbf{if}\ ts'.Id = (GetTS\ SelMatch).Id \\
\qquad\qquad\qquad\qquad \mathbf{then}\ ts.PendFull \setminus \{SelBlocked\} \\
\qquad\qquad\qquad\qquad \mathbf{else}\ ts.PendFull \bullet ts' \,\} \wedge \\
\qquad ActiveProcs' = ActiveProcs \cup \{first\ SelBlocked\}))))
\end{array}
$$

The set *Matchings* contains all the matching tuples. They are collected together with the corresponding tuple space, the template, and the optional statement. One such tuple is then selected, for which the following actions are necessary:

(assign the tuple fields to the l-values in the corresponding formals (optional)) $\wedge$
(execute the associated statements (optional)) $\wedge$
(remove the matching tuple from the tuple space) $\wedge$
(if the tuple space was full and there are processes pending on this full tuple space:
  (remove one of these pending processes from *PendFull*) $\wedge$
  (add the corresponding tuple to the bag *Tuples*) $\wedge$
  (reactivate the pending process))

As you can see, fetching a tuple may reactivate a process that is blocked with a `deposit` operation on a full tuple space.

If no matching tuple has been found, the templates together with the requesting process will be added to the appropriate lists of pending `fetch` templates, provided that no `else` statements are specified:

$$
\begin{array}{l}
\textit{FetchNoMatch} \\
\hline
\Delta \textit{Program} \\
\textit{InTempList?} : \textit{TempList} \\
\textit{InProc?} : \textit{Process} \\
\textit{Else?} : \textit{OptStmt} \\
\hline
\forall \textit{pair} : \operatorname{ran} \textit{InTempList?} \bullet \textit{second pair} \in \textit{IDsOF TSs} \\
\neg\, (\exists \textit{pair} : \operatorname{ran} \textit{InTempList?}; \textit{ts} : \textit{TSs} \mid (\textit{ts.Id} = \textit{second pair}) \bullet \\
\quad \exists \textit{tos} : \operatorname{ran}(\textit{first pair}); \textit{tup} : \operatorname{dom} \textit{ts.Tuples} \bullet \textit{tup}\ \mathsf{Matches}\ (\textit{first tos})) \\
\textit{Else?} = \textit{NoStmt} \\
\textit{TSs}' = \{\, \textit{ts} : \textit{TSs}; \textit{ts}' : \textit{TupleSpace}; \textit{pair} : \operatorname{ran} \textit{InTempList?} \mid \textit{ts}'.\textit{Id} = \textit{ts.Id} \wedge \\
\qquad \textit{ts}'.\textit{Limit} = \textit{ts.Limit} \wedge \\
\qquad \textit{ts}'.\textit{Tuples} = \textit{ts.Tuples} \wedge \\
\qquad \textit{ts}'.\textit{PendMeet} = \textit{ts.PendMeet} \wedge \\
\qquad \textit{ts}'.\textit{PendFull} = \textit{ts.PendFull} \wedge \\
\qquad (\mathbf{let}\ \textit{NewPends} == \{\, \textit{tos} : \operatorname{ran}(\textit{first pair}); \textit{pe} : \textit{Pending} \mid \textit{pe.proc} = \textit{InProc?} \wedge \\
\qquad\qquad\qquad \textit{pe.temp} = \textit{first tos} \wedge \textit{pe.os} = \textit{second tos} \bullet \textit{pe} \,\} \bullet \\
\qquad \textit{ts}'.\textit{PendFetch} = \mathbf{if}\ \textit{ts}'.\textit{Id} = \textit{second pair} \\
\qquad\qquad \mathbf{then}\ \textit{ts.PendFetch} \cup \textit{NewPends} \\
\qquad\qquad \mathbf{else}\ \textit{ts.PendFetch}) \\
\qquad\quad \bullet\ \textit{ts}' \,\} \\
\textit{ActiveProcs}' = \textit{ActiveProcs} \setminus \{\textit{InProc?}\} \\
\hline
\end{array}
$$

If no matching tuple has been found and `else` statements are specified, our program state does not change and the `else` statements are executed:

DoElseStmt

$\Xi Program$
$InTempList? : TempList$
$InProc? : Process$
$Else? : OptStmt$

$\forall pair : \operatorname{ran} InTempList? \bullet second\ pair \in IDsOF\ TSs$

$\neg (\exists pair : \operatorname{ran} InTempList?; ts : TSs \mid (ts.Id = second\ pair) \bullet$
$\quad \exists tos : \operatorname{ran}(first\ pair); tup : \operatorname{dom} ts.Tuples \bullet tup\ \mathsf{Matches}\ (first\ tos))$

$Else? \neq NoStmt$

$OptExecute\ Else?$

The exception `ts_invalid_id` will be raised when invalid tuple-space identities are given:

InvalidTempList

$\Xi Program$
$InTempList? : TempList$
$InProc? : Process$
$Exception! : Statement$

$\neg (\forall pair : \operatorname{ran} InTempList? \bullet second\ pair \in IDsOF\ TSs)$

$Exception! =$ '`escape ts_invalid_id();`'

No `into` expressions are allowed for `fetch` operations:

DisallowIntos

$InTempList? : TempList$

$\forall pair : \operatorname{ran} InTempList? \bullet \forall tos : \operatorname{ran}(first\ pair) \bullet$
$\quad \neg HasIntos(first\ tos)$

We specify this condition in a separate schema to allow the reuse of *DoElseStmt* and *InvalidTempList* for the *Meet* operation (see below).

The disjunction of *FetchMatch*, *FetchNoMatch*, *DoElseStmt*, and *InvalidTempList* in conjunction with *DisallowIntos* then constitutes *Fetch*:

$$Fetch \mathrel{\widehat=} (FetchMatch \lor FetchNoMatch \lor DoElseStmt \lor InvalidTempList) \land DisallowIntos$$

Note that, although processes are unique, a tuple-space operation such as the following would yield two identical pending templates for tuple space `TS`:

```
fetch (1) xor (1) at TS end fetch;
```

But since the set of pending processes with associated templates is a set, only one such pair would be added to the corresponding set in *TupleSpace*: the second added template would replace the first one. However, this produces no problems, because it does not matter which one of such identical templates might be selected, provided one could be selected at all. If statements are specified, the process attributes would be different, and therefore the second added template would not replace the first one (provided that the statements are different). Replacing these sets e.g. by bags would complicate our specification unnecessarily.

2.6.4 Meeting tuples

This section defines the *Meet* operation incrementally. The informal specification is given in section D.2.3. Meeting tuples is similar to fetching tuples except that the matching tuple is not removed from tuple space and that it may be changed via `into` expressions while meeting it:

$$
\begin{array}{l}
\textbf{MeetMatch} \\
\Delta Program \\
InTempList? : TempList \\
InProc? : Process \\
\hline
\forall pair : \operatorname{ran} InTempList? \bullet second\ pair \in IDsOF\ TSs \\[1ex]
\exists pair : \operatorname{ran} InTempList?;\ ts : TSs \mid (ts.Id = second\ pair) \bullet \\
\quad \exists tos : \operatorname{ran}(first\ pair);\ tup : \operatorname{dom} ts.Tuples \bullet tup\ \mathsf{Matches}\ (first\ tos) \\[1ex]
\textbf{let}\ Matchings == \{ ts : TSs;\ pair : \operatorname{ran} InTempList?;\ TEMP : Template; \\
\quad TUP : APTuple;\ OS : OptStmt \mid (\exists tos : \operatorname{ran}(first\ pair);\ tup : \operatorname{dom} ts.Tuples \bullet \\
\quad tup\ \mathsf{Matches}\ (first\ tos) \land TEMP = first\ tos \land TUP = tup \land OS = second\ tos) \\
\qquad \bullet (ts, TEMP, TUP, OS) \} \bullet \\
(\exists SelMatch : Matchings \bullet \\
\quad GetTemp\ SelMatch\ \mathsf{FormalAssign}\ GetTup\ SelMatch \land \\
\quad OptExecute\ (GetOS\ SelMatch) \land \\
\quad (\neg HasIntos(GetTemp\ SelMatch) \Rightarrow \\
\qquad (TSs' = \{ ts : TSs;\ ts' : TupleSpace \mid ts.Id = ts'.Id \land \\
\qquad\qquad ts'.Limit = ts.Limit \land \\
\qquad\qquad ts'.Tuples = ts.Tuples \land \\
\qquad\qquad ts'.PendFetch = ts.PendFetch \land \\
\qquad\qquad ts'.PendMeet = ts.PendMeet \land \\
\qquad\qquad ts'.PendFull = ts.PendFull \bullet ts' \} \land \\
\qquad ActiveProcs' = ActiveProcs)) \land \\
\quad (HasIntos(GetTemp\ SelMatch) \Rightarrow \\
\quad (\exists_1 Program'' \bullet \\
\qquad TSs'' = \{ ts : TSs;\ ts'' : TupleSpace \mid ts''.Id = ts.Id \land \\
\qquad\qquad ts''.Limit = ts.Limit \land \\
\qquad\qquad ts''.Tuples = \textbf{if}\ ts''.Id = (GetTS\ SelMatch).Id \\
\qquad\qquad\qquad\qquad \textbf{then}\ ts.Tuples \uplus [\![GetTup\ SelMatch]\!] \\
\qquad\qquad\qquad\qquad \textbf{else}\ ts.Tuples \land \\
\qquad\qquad ts''.PendMeet = ts.PendMeet \land \\
\qquad\qquad ts''.PendFetch = ts.PendFetch \land \\
\qquad\qquad ts''.PendFull = ts.PendFull \bullet ts'' \} \land \\
\qquad ActiveProcs'' = ActiveProcs \land \\
\qquad \theta Program' = \theta Program''\ \mathsf{AddTuple} \\
\qquad\quad (GetTemp\ SelMatch\ \mathsf{EvalIntos}\ GetTup\ SelMatch, (GetTS\ SelMatch).Id))))
\end{array}
$$

If there exists a template in the given *InTempList?*, which Matches a tuple in the associated tuple space, then the following actions are necessary:

(assign the tuple fields to the l-values in the corresponding formals (optional)) ∧
(execute the associated statements (optional)) ∧
(if there are `intos`:
  (remove the matching tuple from the tuple space) ∧
  (add the changed tuple via AddTuple))

If no `intos` are specified, our program state does not change. If `intos` are specified, the matching tuple has to be removed and the changed one has to be added. The addition of the changed tuple might satisfy other pending processes with templates matching the changed tuple.

If no matching tuple has been found, the templates together with the requesting process will be added to the lists of pending `meet` templates, provided that no `else` statements are specified:

$$
\begin{array}{l}
\textbf{MeetNoMatch} \\
\Delta Program \\
InTempList? : TempList \\
InProc? : Process \\
Else? : OptStmt \\
\hline
\forall pair : \mathrm{ran}\ InTempList? \bullet second\ pair \in IDsOF\ TSs \\
\neg (\exists pair : \mathrm{ran}\ InTempList?; ts : TSs \mid (ts.Id = second\ pair) \bullet \\
\quad \exists tos : \mathrm{ran}(first\ pair); tup : \mathrm{dom}\ ts.Tuples \bullet tup\ \mathsf{Matches}\ (first\ tos)) \\
Else? = NoStmt \\
TSs' = \{ ts : TSs; ts' : TupleSpace; pair : \mathrm{ran}\ InTempList? \mid ts'.Id = ts.Id \land \\
\quad ts'.Limit = ts.Limit \land \\
\quad ts'.Tuples = ts.Tuples \land \\
\quad ts'.PendFetch = ts.PendFetch \land \\
\quad ts'.PendFull = ts.PendFull \land \\
\quad (\mathbf{let}\ NewPends == \{ tos : \mathrm{ran}(first\ pair); pe : Pending \mid pe.proc = InProc? \land \\
\qquad pe.temp = first\ tos \land pe.os = second\ tos \bullet pe \} \bullet \\
\quad ts'.PendMeet = \mathbf{if}\ ts'.Id = second\ pair \\
\qquad \mathbf{then}\ ts.PendMeet \cup NewPends \\
\qquad \mathbf{else}\ ts.PendMeet) \\
\quad \bullet ts' \} \\
ActiveProcs' = ActiveProcs \setminus \{InProc?\}
\end{array}
$$

MeetNoMatch is similar to *FetchNoMatch*, except that *PendMeet* and *PendFetch* are exchanged.

The disjunction of *MeetMatch*, *MeetNoMatch*, *DoElseStmt*, and *InvalidTempList* then constitutes the *Meet* operation:

$$Meet \mathrel{\widehat{=}} MeetMatch \lor MeetNoMatch \lor DoElseStmt \lor InvalidTempList$$

2.7 Execution

Internal computations of processes do not change the program state of our specification. We can model them via the operation *NoOp*:

$$
\begin{array}{l}
\textbf{NoOp} \\
\Xi Program
\end{array}
$$

An execution of a program is therefore an infinite sequence of program state changes, each one related to its immediate successor by the operation schemas defined above. We shall model this by defining an *execution history* as follows:

$$
\begin{array}{|l}
\textit{Execution} \\
\textit{History} : \mathbb{N} \longrightarrow \Delta \textit{Program} \\
\hline
\exists_1 \Delta \textit{Program} \bullet \textit{History}\, 0 = \theta \textit{InitProgram} \\
\forall i : \mathbb{N}_1 \bullet \\
\quad \exists \Delta \textit{Program} \bullet \\
\quad\quad \textit{History}\, i = \theta \Delta \textit{Program} \wedge \\
\quad\quad (\textit{History}\, i).\textit{TSs} = (\textit{History}\,(i-1)).\textit{TSs}' \wedge \\
\quad\quad (\textit{History}\, i).\textit{ActiveProcs} = (\textit{History}\,(i-1)).\textit{ActiveProcs}' \wedge \\
\quad\quad (\, \textit{NoOp} \vee \\
\quad\quad \textit{ProgramTermination} \vee \\
\quad\quad (\exists \textit{ProcessCreation}; p : \textit{Process} \bullet p = \textit{NewProcess}?) \vee \\
\quad\quad (\exists \textit{ProcessTermination}; p : \textit{Process} \bullet p = \textit{ToKill}?) \vee \\
\quad\quad (\exists \textit{CreateTS}; v : \textit{Value} \bullet v = \textit{InLimit}?) \vee \\
\quad\quad (\exists \textit{ExistsTS}; v : \textit{Value} \bullet v = \textit{InTS}?) \vee \\
\quad\quad (\exists \textit{ClearTS}; v : \textit{Value} \bullet v = \textit{InTS}?) \vee \\
\quad\quad (\exists \textit{RemoveTS}; v : \textit{Value} \bullet v = \textit{InTS}?) \vee \\
\quad\quad (\exists \textit{Deposit}; \textit{tl} : \textit{TupList}; b : \textit{BlockMode}; p : \textit{Process} \bullet \\
\quad\quad\quad \textit{tl} = \textit{InTupList}? \wedge b = \textit{Blocking}? \wedge p = \textit{InProc}?) \vee \\
\quad\quad (\exists \textit{Fetch}; \textit{tl} : \textit{TempList}; e : \textit{OptStmt}; p : \textit{Process} \bullet \\
\quad\quad\quad \textit{tl} = \textit{InTempList}? \wedge e = \textit{Else}? \wedge p = \textit{InProc}?) \vee \\
\quad\quad (\exists \textit{Meet}; \textit{tl} : \textit{TempList}; e : \textit{OptStmt}; p : \textit{Process} \bullet \\
\quad\quad\quad \textit{tl} = \textit{InTempList}? \wedge e = \textit{Else}? \wedge p = \textit{InProc}?)\,) \\
\forall \Delta \textit{Program}; i : \mathbb{N}_1 \bullet \\
\quad \textit{History}\, i = \theta \textit{ProgramTermination} \Rightarrow (\forall j : \mathbb{N}_1 \mid j > i \bullet \textit{History}\, j = \theta \textit{NoOp}) \\
\hline
\end{array}
$$

Z A schema reference may be used as an expression: its value is the set of bindings in which the values of the components obey the property of the schema. Z

For any given program, there is therefore a set of valid execution histories. The first state change in the sequence represents the initialization. It is not required that programs terminate. However, they may terminate: after *ProgramTermination* only *NoOps* are allowed. We model terminating programs via infinite sequences to keep the specification simple.

We presented an operational semantics which defines the possible state changes of our *Program* state through the operations on this state. We model concurrency as usual when operational semantics are used:

> "As usual when parallelism is specified by an operational semantics, concurrency is decribed by an arbitrary interleaving of a set of atomic transactions performed by the acting processes [Milner, 1989].
>
> The interleaving approximation does not provide true concurrency, but it is sufficient as tuples are indivisible units which are manipulated by atomic actions. ... The possibility of blocking operations are modeled by the *match* relation: ..."
> [Ciancarini *et al.*, 1992, page 7]

Note that the notions of distribution and asynchrony are not captured by such an operational semantics. The goal of the presented work is not to specify as much parallelism as possible. The goal is to provide a precise specification of the semantics of generative communication in ProSet.

2.8 Fairness

The reader is referred to section D.2.4 for an informal discussion of weakly fair selection. We specify weakly fair selection of processes that are pending for templates via the following global property for

execution histories:

$$\begin{array}{l}
\forall\, Execution;\ i : \mathbb{N};\ ts : TupleSpace;\ pp : Pending \mid ts \in (History\ i).TSs\ \wedge \\
\qquad\qquad\qquad\qquad pp \in (ts.PendFetch \cup ts.PendMeet) \bullet \\
\quad \neg\, (\exists\, card : \mathbb{N} \bullet card = \#\{\, j : \mathbb{N};\ ots : TupleSpace \mid ots \in (History\ j).TSs \wedge ots.Id = ts.Id\ \wedge \\
\qquad\qquad\qquad\qquad (\exists\, tup : \mathrm{dom}\ ots.Tuples \bullet tup\ \mathsf{Matches}\ pp.temp) \bullet j\,\}) \\
\qquad \Rightarrow \\
\quad (\exists\, k : \mathbb{N};\ ots : TupleSpace \mid k > i \wedge ots.Id = ts.Id \wedge ots \in (History\ k).TSs\ \bullet \\
\qquad pp \notin (ots.PendFetch \cup ots.PendMeet))
\end{array}$$

We use the name *ots* as an acronym for *other tuple spaces*. An informal outline of this predicate follows:

(forall *Execution*s where a *TupleSpace* in (*History i*) contains a pending `meet` or `fetch`:
(there exists infinitely often a matching tuple in the *Execution*)
$\Rightarrow$
(this pending `meet` or `fetch` will be selected sometime after (*History i*)))

We check that a set is finite in the following way. The predicate

$$\neg\, (\exists\, card : \mathbb{N} \bullet card = \#\{\ldots\})$$

provides *true* if $\{\ldots\}$ is an infinite set. The set comprehension in the above property contains the indices to the history components, which contain matching tuples for the template of pending process *pp*. If this is an infinite set, this templates has to be selected sometime.

We specify weakly fair selection of processes that are pending for full tuple spaces in a similar way:

$$\begin{array}{l}
\forall\, Execution;\ i : \mathbb{N};\ ts : TupleSpace;\ b : Process \times APTuple \mid \\
\qquad\qquad\qquad\qquad ts \in (History\ i).TSs \wedge b \in ts.PendFull \bullet \\
\quad \neg\, (\exists\, card : \mathbb{N} \bullet card = \#\{\, j : \mathbb{N};\ ots : TupleSpace \mid ots \in (History\ j).TSs \wedge ots.Id = ts.Id\ \wedge \\
\qquad\qquad\qquad\qquad IntValueOf\ ots.Limit > BagSum\ ots.Tuples \bullet j\,\}) \\
\qquad \Rightarrow \\
\quad (\exists\, k : \mathbb{N};\ ots : TupleSpace \mid k > i \wedge ots \in (History\ k).TSs \wedge ots.Id = ts.Id\ \bullet \\
\qquad b \notin ots.PendFull)
\end{array}$$

If you replace matching tuples by non-full tuple spaces in the above informal outline, this may also help to understand the last formula.

In section 3.4 we shall sketch how to design a workable implementation for our specification of weakly fair selection of pending templates.

3 Correctness of the design for an implementation

Once a semantics of tuple space has been given, it is obvious to consider the properties of an implementation design, too. Obtaining a design for an implementation from a specification may be done via top-down, step-wise data and operation refinement. Data refinement is the process of transforming one data type into another one: an abstract data type is transformed into a more concrete one. It is possible to carry out data refinement in Z by using appropriate schemas. In order to prove that the design is a correct refinement of the specification we have to prove a number of things. Firstly we have to prove that the initial states correspond to one another and then — for each operation in the design — we have to prove that it is both correct and also that it is applicable.

3.1 Semantics vs. implementation design

The semantics of the specification is called the *universal semantics*, since it is the union of all valid semantics of possible implementation designs. No real implementation design is likely to implement this universal semantics, however, this does not mean that it is not a valid interpretation of the semantics. So the design for any implementation which is a subset of the universal semantics is a valid interpretation of the specification. What this means is that if a program is proven correct according to the universal semantics, and then run on a system which has a valid interpretation of those semantics, then the program will run correctly. The distinction between universal semantics and valid interpretations is a useful framework for the validation of implementation designs.

The core of the above is that an implementation design is correct if it enables a subset of the action sequences, leading to a subset of the possible blocked states, deducible from the specification. It should be noted that by this property an implementation design will be correct, even if it for a certain program state only enables a subset of the set of actions deducible from the specification, i.e. the implementation design need not enable every operation deducible from the specification.

We can relate our specification of a program to an implementation design, and hereby provide a correctness relation one such design must satisfy. The property of this relation specifies that the specification must be able to simulate an implementation design, i.e. trace all its actions. Furthermore the implementation may only block whenever the specification says so. Whatever an implementation does in a certain configuration, it must also be possible to do in the context of the specification. Furthermore, if no actions may be deduced in the context of the specification, the implementation must block, too.

In summary, if a certain behavior of a program is deducible for all computations within the specification, the implementation design must show the same behavior of the program. Conversely, if nondeterministic behavior may be deduced from the specification, the implementation design need not reflect this; a fully determinate behavior is considered correct.

An underlying problem in the correctness of an implementation design is scheduling: it is of course undesirable to demand that an implementation has all the nondeterminism inherent in the specification. That is, we will at most demand that the set of possible behaviors of an implementation design is a subset of the possible behaviors of the specification, a subset which must fulfill two requirements to be considered correct:

liveness If a set of processes is able to act in the context of the specification, then at least one of them must be able to do so in the context of the implementation design, too. The implementation design must have a subset of the liveness properties of the specification. This implies that whenever a process in the context of the specification is guaranteed to terminate, the process also terminates in the context of the design.

deadlock If no process can act in the context of the specification, then they neither may be able to do so in the context of an implementation design. The specification and an implementation must have the same deadlock properties.

Given that the tuple space operations will not be atomic in an implementation — they will be transactions composed by some low-level atomic actions of the specific implementation — these demands become problematic: exactly when do the tuple space operations take effect in an implementation, and can they be ordered such that they satisfy the liveness properties. An interesting aspect in correctness of an implementation design is how to deal with different granularity of atomicity and to provide a correctness relation which bridges this.

3.2 Optimizations

If optimizations are introduced, a model of the optimizations could be developed and incorporated into the existing model. This could then prove that the optimizations were safe, in the sense that they do not alter the semantics of the specification.

3.3 Finite computer systems

The specification of program states implies some unboundness, which an implementation with limited resources somehow has to relax. A restriction, which may be imposed by an implementation, is e.g. an upper bound of active processes. Thus an implementation will never be able to satisfy the correctness relation fully, only a weaker property: given a set of implementation constraints, and a set of programs not violating them, then the implementation is correct with respect to these programs.

3.4 Fairness

Let us now consider the implementation design of the fairness properties specified in section 2.8 as an example for refinement. We only sketch the design in the present paper.

We say that a given implementation design of our specification is *correct* if the set of possible execution histories of a particular program under the implementation design is a subset of the execution histories of that program. Note that, in general, there is no means of determining exactly what this subset will be for any particular implementation design.

For data refinement an abstraction relation between the abstract state space of the specification and the concrete state space of an implementation design has to be given. Operation refinement leads to algorithm development. The before and after states of operations of an implementation design must provide before and after states of the corresponding operations which are related to each other. What we require is that any program which is a correct implementation of the design is also a correct implementation of the specification. The operations in the design *model* those in the specification. See e.g. [Spivey, 1992b, sections 1.5 and 5.6] for a simple example.

The specified weak fairness property for pending processes in the specification could be implemented by using FIFO queues for pending processes instead of sets as in *TupleSpace* (section 2.3). In a FIFO (first-in first-out) queue always the possible candidate who waits longest is selected first. The design for tuple spaces then could use injective sequences for pending processes:

TupleSpaceDesign

$Id : Value$
$Limit : Value$
$Tuples : \mathrm{bag}\ APTuple$
$PendFetchDesign : \mathrm{iseq}\ Pending$
$PendMeetDesign : \mathrm{iseq}\ Pending$
$Blocked : Process \pinj APTuple$

$(Type\ Id = atom) \land (Id \notin \mathrm{dom}\ ValuesOfType)$

$(Type\ Limit = integer) \lor (Type\ Limit = om)$

Z iseq X is the set of injective finite sequences over X: these are precisely the finite sequences over X which contain no repetitions. Z

Similar changes may be done for the other tuple-space components. The matching procedure would select matching templates according to their position in the sequences. Also multiple sequences for pending processes, separated with respect to possible matching tuple sets, may be useful. The specification does not restrict us to a particular implementation design (as long as it is a valid interpretation of the specification). An implementation according to the design given above restricts the nondeterminism possible in the specification, but still prevents starvation.

We can document the correspondence between specification and design with a schema *Abstraction* that defines the *abstraction relation* between the abstract state space *TupleSpace* and the concrete state space *TupleSpaceDesign*:

$$\begin{array}{l}
\textit{Abstraction} \\
\hline
\textit{TupleSpace} \\
\textit{TupleSpaceDesign} \\
\hline
\forall pf : \textit{PendFetch}; pfd : \operatorname{ran} \textit{PendFetchDesign} \bullet \\
\quad pf \in \operatorname{ran} \textit{PendFetchDesign} \land pfd \in \textit{PendFetch} \\
\forall pm : \textit{PendMeet}; pmd : \operatorname{ran} \textit{PendMeetDesign} \bullet \\
\quad pm \in \operatorname{ran} \textit{PendMeetDesign} \land pmd \in \textit{PendMeet}
\end{array}$$

Having explained what the concrete tuple-space design is, and how concrete and abstract state spaces are related, we can begin to give designs for the operations of the specification. It is necessary to prove that the operations for the concrete state space are correct implementations of the operations for the abstract state space, and that the global properties, such as fairness, hold. However, this is subject for further work.

4 Conclusions

A formal specification of ProSet-Linda has been presented by means of the formal specification language Z. The specification of the formal semantics of generative communication in ProSet led us to the recognition of several omissions and imperfections in our previous informal specification. Nevertheless, the main advantage of using Z lies in subsequent developments of this work. Changes in the language can be analyzed by formal transformations of its current specification. Implementations can be formally derived, different strategies can be identified and choices for optimizations can be well motivated and documented. Z specifications could be informally developed into programs, which gives the implementor/user more confidence in the final system. There may be bugs, but they are less likely to be at the conceptual level. Formal development involves some sort of transformation/refinement. Refinement is usually regarded as a step-wise approach, possibly with proof obligations for each step. Such development may be aided by tools. Many people regard this method of development as tedious, but it allows the developer to have confidence in the end product (provided the specification and development method are "correct"). A fully formal development process is more expensive in terms of time and education, but much cheaper in terms of maintenance.

However, while constructing the presented formal specification in Z we recognized one important drawback of using Z with the *f*UZZ package [Spivey, 1992a]: the principle of *definition before use*, which leads to a bottom-up development of the specification. It would be more natural to write and explain the specification in a top-down manner. The important point is just that any specification must be written in a way such that its definitions can be ordered to satisfy the principle of definition before use [Spivey, 1992b, page 47]. This avoids recursive definitions in which a schema includes itself, for instance. Therefore, it should be possible to develop a tool for Z that allows the introduction of paragraphs in any order and ensures that the principle of definition before use can be satisfied. It would be nice to have a *f*UZZ directive, which *announces* a forthcoming definition. The existing *f*UZZ directives allow preliminary, invisible definitions, but the later final definitions cannot be type-checked, because they are redefinitions of global names.

There exist also some proposals for object-oriented extensions to Z. However, their goal is not to overcome the restriction to a bottom-up development of Z specifications. E.g., the goal of the MooZ specification language [Meira and Cavalcanti, 1992] is to provide object-oriented structuring facilities to Z so that the specification of large software systems can be constructed and managed more adequately, but still employs the principle of definition before use. The object-oriented paradigm promotes the use of a bottom-up development in which the whole system is constructed from its component parts.

Concurrency is described by arbitrary interleavings of the atomic actions of the participating processes. However, nothing in the semantics given here prevents causal independent actions to occur in parallel.

The concurrency of programs is modeled by the nondeterministic interleaving of atomic actions, i.e. an asynchronous model. Atomic transitions happen one after another in a non-fixed arbitrary order. The interleaving semantics defined by the execution history suggests a centralized implementation, because the execution history has a unique thread of control, but if no local configuration is shared by two configurations, they have no way to communicate and thereby, no possible interference arises from a parallel occurrence of their transitions. Our work provides a distributed semantics since there is no global state in our (global) configurations: local configurations for multiple tuple spaces are specified instead. The overall transition of programs is specified in terms of individual process transitions: a transition is based on a partial view of tuple spaces. This rule reflects the local decision making property of Linda: the transitions involve only small subsets of tuples. This local decision making property is the source for time and space decoupling of processes, which makes programming in Linda so easy compared to other paradigms for coordinated programming such as message passing.

Weakly fair selection of pending processes has been specified for *execution histories* and we sketched the design for a workable implementation for it. [Dijkstra, 1988] argues that “fairness, being an unworkable notion, can be ignored with impunity” because finite experiments cannot distinguish between fair and unfair implementations. Conversely, we view fairness as a simplifying assumption to increase abstraction. Concepts that cannot be verified by finite experiments, such as liveness properties, are introduced to make program design simpler. Fairness is such a liveness property, which allows us — among other things — to reason about program termination in the presence of nondeterminism. Another approach to abstraction is the use of real numbers to specify numerical calculations: even though they cannot be represented exactly by computers, real numbers provide a convenient language for describing calculations which the computer will carry out approximately.

We specified the conditions under which exceptions have to be raised. The actions to be taken to handle such situations were only sketched. This is due to the fact that the present paper only presents a specification of generative communication in ProSet, and not a specification of the entire language. A rigorous formal specification of exception handling is in general not a light-weight exercise.

The concept for process creation in ProSet is adapted from Multilisp's futures. The resolving and touching of futures is only specified for processes within active tuples in tuple space, and not for processes spawned outside of tuple space. This limitation is also due to the fact that this paper is not a specification of the entire language.

We sketched how to develop an implementation design for the presented specification. The goal is to verify that the implementation meets the specification. Note that an implementation, which is considered to be correct, cannot be claimed to be fully reliable. An implementation design that has been verified is *not* immune from bugs, although the probability that it contains bugs is very much smaller than if it had not been verified. We want reliability and not perfection. Writing a proof is somewhat like writing a program, and is subject to error in the same ways that programs are subject to error. Nevertheless, a machine checked proof does give us a very high degree of confidence in the correctness of a program, even though it cannot guarantee total reliability. Some chance of failure always remains, no matter how remote. [DeMillo *et al.*, 1979] argue that believing a proof is a social process and that proofs consisting entirely of calculations are not necessarily correct. A proof for a refinement step is a *message* to the community, which says that we *believe* it is correct. Therefore, our goal is not to design a fully reliable implementation — we even do not believe that this would be possible in practice. Our goal is to gain high confidence in the design for an implementation.

ProSet and Z appear to be a good combination for software engineering in general. Prototyping is not meant to replace the entire software life cycle model. It is intended as a completion. Because of the similarities between ProSet and Z it may be a good idea to build an executable prototype — derived from a specification written in Z — in ProSet. Presenting an executable prototype for a specification is often called *specification animation*. In [Diller, 1990], prototypes for Z specifications were constructed in Miranda and Prolog, but set-oriented programming techniques may be a more adequate choice for constructing prototypes from Z specifications than functional or logic programming techniques.

A The prototyping language ProSet

This appendix provides a brief introduction to data and control structures of the prototyping language ProSet. For a full account see [Doberkat *et al.*, 1992]. The high-level structures that ProSet provides qualify the language for prototyping.

A.1 Data structures

ProSet provides the first-class data types **atom**, **integer**, **real**, **string**, **boolean**, **tuple**, **set**, **function**, **modtype**, and **instance**. It is a *higher-order* language, because functions and modules have first-class rights, too. First-class means to be expressible without giving a name. It implies having the right to be anonymous, being storable in variables and in data structures, being comparable for equality with other objects, being returnable from or passable to a procedure. Each variable or constant is meant to be an *object* for our terminology. ProSet is weakly typed, i.e. the type of an object is in general not known at compile time. Integer, real, string and boolean objects are used as usual. Atoms are unique with respect to one machine and across machines. They can only be created and compared. The unary **type** operator returns a predefined type atom corresponding to the type of its operand.

Atom, integer, real, string, boolean, function and module objects are basic, because they are not built from other objects. Tuples and sets are compound data structures, which may be heterogeneous composed of basic data objects, tuples, and sets. Sets are unordered collections while tuples are ordered. The following expression creates a tuple consisting of an integer, a string, a boolean and a set of two reals:

```
[123, "abc", true, {1.4, 1.5}]
```

Such expressions are called *tuple former*. Sets consisting only of tuples of length two are called maps. There is no genuine data type for maps, because set theory suggests handling them this way. The following statement assigns a set that is a map to the variable `M`:

```
M := { [1,"s1"], ["s2",2], [3,"s3"] };
```

Now the following equalities hold:

```
domain(M) = {1, "s2", 3}
 range(M) = {"s1", 2, "s3"}
     M(1) = "s1"
     M{1} = {"s1"}
```

Domain and range of a map may be heterogeneous. `M{1}` is the *multi-map selection* for relations.

There is also the undefined value **om** which indicates e.g. selection of an element from an empty set. **om** itself may not be element of a set, but of a tuple.

A.2 Control structures

The control structures show that the language has ALGOL as one of its ancestors. There are **if**, **case**, **loop**, **while** and **until** statements as usual and in addition some structures that are custom tailored to the compound data structures. First have a look at expressions forming tuples and sets:

```
T := [1 .. 10];
S := {2*x: x in T | x > 5};    -- result: {12, 14, 16, 18, 20}
```

The iteration "`x in T`" implies a loop in which each element of the tuple `T` is successively assigned to `x`. The visibility of `x` is bound to the set former. For all elements of `T`, which are satisfying the condition "`x > 5`" the result of the expression "`2*x`" is added to the set. As usual in set theory "`|`" means *such that*. With this knowledge the meaning of the following **for** loop should be obvious:

```
for x in S | x > 15 do <statements> end for;
```

The iteration proceeds over a copy, which is created at first. The quantifiers ($\exists$, $\forall$) of predicate calculus are provided, e.g.:

```
if exists x in S | p(x) then <statements> end if;
```

Additionally ProSet provides the **whilefound** loop:

```
whilefound x in S | p(x) do <statements> end whilefound;
```

The loop body is executed provided an existentially quantified expression with the same iterator would yield **true**. The bound variables are local to the **whilefound** loop as they are in **for** loops and in quantified expressions. Unlike **for** loops the iterator is reevaluated for every iteration.

A.3 An example

In Fig. 2 a solution for the so-called *queens' problem* is given to provide a first impression of the language. Informally, the problem may be stated as follows:

> *Is it possible to place N queens ($N \in \mathbb{N}$) on an $N \times N$ chessboard in such a way that they do not attack each other?*

Anyone familiar with the basic rules of chess also knows what "attack" means in this context: in order to attack each other, two queens are placed in the same row, the same column, or the same diagonal.

The program in Fig. 2 does not solve the above problem directly. It prints out the set of all positions in which the `N` queens do not attack each other. If it is not possible to place N queens in non-attacking positions, this set will be empty. We denote positions on the chessboard by pairs of natural numbers for convenience (this is unusual in chess, where characters are used to denote the columns). `[1,1]` denotes the lower left corner. The program in Fig. 2 with `N=4` produces the following set as a result:

```
{{[1, 3], [2, 1], [4, 2], [3, 4]},
 {[3, 1], [1, 2], [2, 4], [4, 3]}}
```

As sets are unordered collections, the program may print the fields and positions in different sequences. Note that there are no explicit loops and that there is no recursion in the program. All iterations are done implicitly. One may regard this program also as a specification of the queens' problem.

```
program Queens;
   constant N := 4;
begin
   fields := {[x,y]: x in [1..N], y in [1..N]};

   put ({NextPos: NextPos in npow(N, fields) | NonConflict(NextPos)});

   procedure NonConflict (Position);
   begin
      return forall F1 in Position, F2 in Position |
                  ((F1 /= F2) !implies
                     (F1(1) /= F2(1) and F1(2) /= F2(2) and
                     (abs(F2(1)-F1(1)) /= abs(F2(2)-F1(2)))));
   end NonConflict;

   procedure implies (a, b);
   begin
      return not a or b;
   end implies;

   procedure abs (i);
   begin
      return if i >= 0 then i
                        else -i
             end if;
   end abs;
end Queens;
```

Figure 2: The queens' problem. `npow(k, s)` yields the set of all subsets of the set `s` which contain exactly `k` elements. `NonConflict` checks whether the queens in a given position do not attack each other. It is possible to use procedures with appropriate parameters as user-defined operators by prefixing their names with the "`!`" symbol. This is done here with the procedure `implies`. `T(i)` selects the $\mathtt{i}^{th}$ element from tuple `T`.

B The specification language Z

This appendix provides a brief informal introduction to the formal specification language Z. For a full account we refer to [Diller, 1990]. [Spivey, 1992b] is the de-facto standard for Z. The present appendix and the notes, which are enclosed in the symbols $\underline{\boxed{Z}}$ and $\overline{Z}|$, are derived from these texts.

A Z specification document consists of interleaved passages of formal, mathematical text and informal prose explanation. The formal text consists of a sequence of paragraphs which gradually introduces the schemas, global variables and basic types of the specification. Each paragraph builts on the ones which come before it (definition before use).

Types in Z are sets: every mathematical expression which appears in a Z specification is given a *type* determining a set known to contain the value of the expression. Each variable is given a type by its declaration. The *basic types* or *given sets* of a specification have no internal structure of interest. A given set may serve to the purpose of abstraction or generality. An object of the real world that does not need to be given a model at a particular abstraction level can be represented by a given set. The predefined basic types are $\mathbb{N}$ and $\mathbb{Z}$. $\mathbb{N}$ is the set of natural numbers $\{0, 1, 2, ...\}$, and $\mathbb{Z}$ is the set of integers. Basic type definitions introduce new basic type names, which become part of the global vocabulary of basic types. They are introduced as follows:

$$[PERSON, IDENTITY]$$

Such a basic type definition introduces one or more basic types. These names must not have a previous global declaration, and their scope extends from the definition to the end of the specification. From these *atomic objects*, composite objects can be put together in various ways. These composite objects are the members of composite types put together with the type constructors of Z. There are three kinds of composite types: set types, Cartesian product types (tuples), and schema types.

An *abbreviation definition* introduces a new global constant, which may later be used as an abbreviation for the specified expression. The following example introduces the name *BIJECTION* as an abbreviation for the set of bijections from *PERSON* to *IDENTITY*:

$$BIJECTION == PERSON \rightarrowtail\!\!\!\!\twoheadrightarrow IDENTITY$$

All kinds of functions, such as bijections, are relations with appropriate implicit constraints. Relations are sets of pairs. Pairs are tuples with two components.

The formal part of a Z specification makes use of two-dimensional graphical constructs for schemas, axiomatic descriptions, and generic descriptions.

B.1 Schemas

Schemas provide a means for structuring specifications. They can be used to describe both the static aspect of a system (the state space and invariant relations on the state), and the dynamic aspects (the operations which change the state). The general form of schemas in Z is as follows:

```
┌─ S ────────────────────────────
│ D
├──────────
│ P
└────────────────────────────────
```

where S is the name of the schema, D is a declaration and P a predicate. D is also called the *signature* of S. For instance the declaration $x, y : \mathbb{Z}$ in a schema introduces the variables x and y with type $\mathbb{Z}$, which are then called the *components* of this schema. Such variables are local to the respective schema, unless the schema is included elsewhere. A schema is included through using its name as a declaration. The component names are then visible within the scope of the respective declaration. P is also called the *property* of the schema. When multiple schemas are combined, then their signatures are joined, and their properties are combined accordingly. The schema name S is global.

There are some standard decorations for names used in describing operations: $'$ for labeling the final state of an operation, ? for labeling its inputs, and ! for labeling its outputs. If we decorate a schema name, this means a copy of this schema in which all the component names have been decorated accordingly.

It is possible to have generic schemas:

```
┌─ S[X_1, ..., X_n] ─────────────
│ D
├──────────
│ P
└────────────────────────────────
```

where the X_i are the formal generic parameters which can occur in the types assigned to the identifiers in the declaration D. Later when the generic object is used, *actual generic parameters* (set-valued) are supplied. These determine the sets which the formal parameters take as their values. The above generic schema might be instantiated as follows:

$$I \mathrel{\widehat=} S[A_1, \ldots, A_n]$$

where the A_i are the actual generic parameters (set-valued) and I is the instantiated schema. New schemas may also be defined this way (via $\widehat=$) by combining old ones with the operations of the *schema calculus* [Spivey, 1992b]. For example, the effect of the schema $\vee$ operator is to make a schema in which the predicate part is the result of joining the predicate parts of its arguments with the logical connective $\vee$.

The schema $\Delta State$ is implicitly defined as the combination of the *before*-state $State$ and the *after*-state $State'$ whenever a schema $State$ is introduced:

```
┌─ ΔState ───────────────────────
│ State
│ State'
└────────────────────────────────
```

This implicit definition may be overridden by explicit definitions. The schema $\Xi State$ is implicitly defined as the state space of a data type whenever a schema $State$ is introduced:

```
┌─ ΞState ───────────────────────
│ State
│ State'
├──────────
│ θState' = θState
└────────────────────────────────
```

Such implicit definitions may be overridden by explicit definitions. $\theta State$ is the binding formation of values to the components of the schema $State$. Let the components of $State$ be $x_1, ..., x_n$. The following law holds:

$$\theta State' = \theta State \Leftrightarrow x_1' = x_1 \wedge ... \wedge x_n' = x_n$$

A state schema groups together variables and defines the relationship that holds between their values. At any instant, these variables define the state of the system which they model. An operation schema defines the relationship between the *before* and *after* states corresponding to one or more state schemas.

B.2 Axiomatic descriptions

The general form of an axiomatic description in Z is as follows:

```
│ D
├──────────
│ P
```

where D is a declaration which introduces one or more global variables and P is an optional predicate that constrains the values that can be taken by the variables introduced in D. The variables declared in D cannot have been previously declared globally and their scope extends to the end of the specification. D becomes part of the *global signature* of the specification and P contributes to a *global property*.

A predicate may appear on its own as a paragraph; it specifies a constraint on the values of previously declared global variables. The effect is as if the constraint had been stated as part of the axiomatic descriptions in which the variables were introduced. Therefore, the predicate P in the above axiomatic description box could also be placed behind this box without changing the semantics.

B.3 Generic descriptions

The general form of a generic description in Z is as follows:

$$[X_1, \ldots, X_n]$$
$$D$$
$$P$$

where the X_i are the formal generic parameters which can occur in the types assigned to the identifiers in the declaration D. The variables declared in D cannot have been previously declared globally and their scope extends to the end of the specification. The predicate P constrains the identifiers introduced in D.

Generic descriptions may be used to define concepts such as relations, functions, sequences, bags and the operations on them.

B.4 Free type definitions

A new basic type may also be introduced by a *free type definition*, where recursive structures are allowed. A free type definition such as

$$T ::= Constant \mid PtoT \langle\langle PERSON \rangle\rangle \mid next \langle\langle T \rangle\rangle$$

is equivalent to the basic type definition

$$[T]$$

extended by the axiomatic description

$$Constant : T$$
$$PtoT : PERSON \rightarrowtail T$$
$$next : T \rightarrowtail T$$
$$\langle \{Constant\}, \operatorname{ran} PtoT, \operatorname{ran} next \rangle \ \mathsf{partition} \ T$$

$X \rightarrowtail Y$ is the set of total injections from X to Y. Finite sequences are enclosed in $\langle$ and $\rangle$. Sets are enclosed in $\{$ and $\}$. ran yields the range of a relation. The left-hand operand of partition has to be an indexed family of sets. partition holds when all these sets are disjoint and when their union is equal to the right-hand operand. ran and partition are keywords in Z. A particular common example of an indexed family of sets is a sequence of sets, which is at base only a function defined on a subset of $\mathbb{N}$.

Note that the device of defining free type definitions adds nothing to the power of Z. It is a convenient shorthand.

B.5 Expressions

The expressions within the boxes and in global constraints are based on first-order logic and set theory. Sets, tuples (with at least two components), and bindings of values to components of schemas are fundamental to Z. Relations, functions, bags (multi-sets), and finite sequences belong to the basic mathematical tool-kit of Z. Sequences may be empty. Here, we only present an example for a set-forming expression in Z:

$$\{ x : \mathbb{N} \mid x \leq 10 \bullet x * x \}$$

wich corresponds to

```
{ x * x:  x in ℕ | x <= 10 }
```

in ProSet. However, this is not a legal expression in ProSet, since $\mathbb{N}$ is an infinite set. Note that compound objects in Z have to be homogeneous, whereas compound objects in ProSet may be heterogeneously composed. The type of the above Z expression is "set of integers" and the type of the above ProSet expression is simply "set". The bound variables are local to the respective constructs in both languages.

C The coordination language Linda

This appendix provides a brief introduction to the coordination language Linda.[1] For a full account to Linda see [Carriero and Gelernter, 1990]. Linda is a coordination language concept for explicitly parallel programming in an architecture independent way, which has been developed by David Gelernter at Yale University [Gelernter, 1985]. Communication in Linda is based on the concept of tuple space, i.e. a virtual common data space accessed by an associative addressing scheme.

Process communication and synchronization in Linda is reduced to concurrent access to a large data pool, thus relieving the programmer from the burden of having to consider all process inter-relations explicitly. The parallel processes are decoupled in time and space in a very simple way. This scheme offers all advantages of a shared memory architecture, such as anonymous communication and easy load balancing. It adds a very flexible associative addressing mechanism and a natural synchronization paradigm and at the same time avoids the well-known access bottleneck for shared memory systems as far as possible.

The shared data pool in the Linda concept is called *tuple space*. Its access unit is the tuple, similar to tuples in ProSet (appendix A). Tuples live in tuple space which is simply a collection of tuples. It may contain any number of copies of the same tuple: it is a multi-set, not a set. Tuple space is the fundamental medium of communication. Process communication and synchronization in Linda is also called *generative communication*, because tuples are added to, removed from, and read from tuple space concurrently. Synchronization is done implicitly.

Reading access to tuples in tuple space is associative and not based on physical addresses — in fact, the internal structure of tuple space is hidden from the user. Reading access to tuples is based on their expected content described in so-called *templates*. This method is similar to the selection of entries from a data base. Each component of a tuple or template is either an *actual*, i.e. holding a value of a given type, or a *formal*, i.e. a placeholder for such a value. A formal is prefixed with a question mark. Tuples in tuple space are selected by a matching procedure, where a tuple and a template are defined to match, iff they have the same structure (corresponding number and type of components) and the values of their actuals are equal to the values of the corresponding tuple fields.

C-Linda defines six operators, which may be added to a sequential computation language. These operators enable sequential processes, specified in the underlying computation language, to access the tuple space:[2]

`out(tuple);` The specified tuple is evaluated and then added to the tuple space. The `out`-executing process continues as soon as evaluation of the tuple is completed. The "`out("data", 123);`" operation in Fig. 3 deposits the tuple `["data",123]` into tuple space.

`eval(tuple);` Executing an `eval` operation causes the following sequence of activities. First, bindings for names indicated explicitly in the tuple are established in the environment of the `eval`-executing process. At this point, the `eval`-executing process may continue. Each field of the tuple argument to `eval` is now evaluated, independently of and asynchronously with the `eval`-executing process and each other. The fields of an `eval` tuple are evaluated concurrently yielding one thread of execution for every field. `eval` deposits *active* tuples into tuple space, which are **not** accessible to the remaining four operations. Conversely, `out` deposits *passive* tuples into tuple space, which are accessible to the remaining four operations, which are discussed below. When every field has been evaluated completely, the tuple consisting of the values yielded by each `eval`-tuple field, in the order of their appearance in the `eval` tuple, becomes available in tuple space: the active tuple converts to a passive one. The "`eval("p",p());`" operation in Fig. 3 deposits the active tuple `["p",p()]`, containing two processes, into tuple space.

The main program is the only process that lives outside of tuple space.

[1] Linda is a registered trademark of Scientific Computing Associates.

[2] Parts of this description were derived from [Carriero and Gelernter, 1990, appendix A].

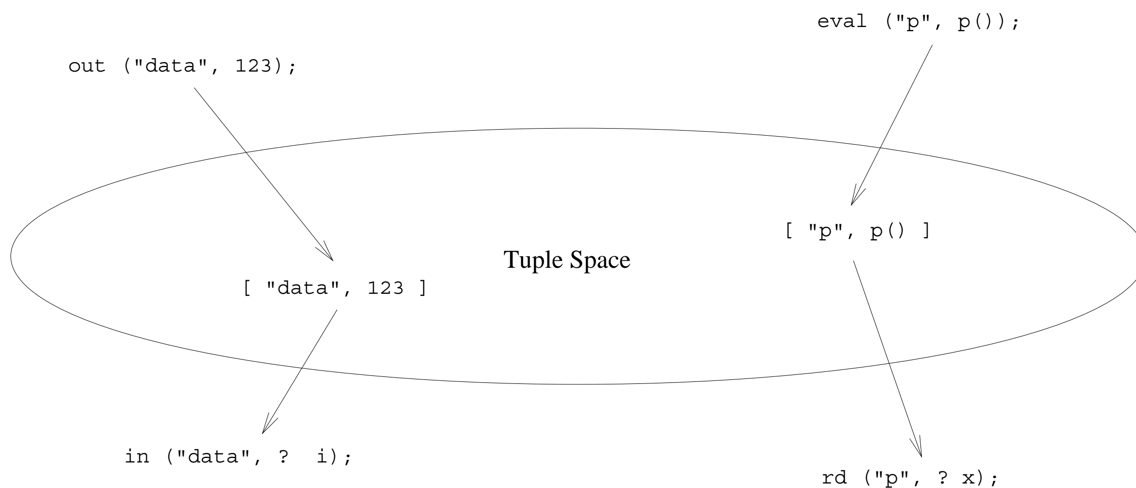


Figure 3: Tuple-space communication in C-Linda.

`in(template);` The `in` operation attempts to withdraw a specified tuple from tuple space. Tuple space is searched for a matching tuple against the template supplied as the operation's argument. When and if a tuple is found, it is withdrawn from tuple space, and the values of its actual fields are bound to any corresponding formals in the template. Tuples are withdrawn *atomically*: a tuple can be grabbed by only one process, and once grabbed it is withdrawn entirely. If no matching tuple exists in tuple space, the process executing the `in` suspends until a matching tuple becomes available. If many tuples satisfy the match criteria, one is chosen arbitrarily. The "`in("data",?i);`" operation in Fig. 3 withdraws the tuple `["data",123]` from tuple space and assigns `123` to the integer variable `i`.

`rd(template);` The `rd` operation is the same as `in`, with actuals assigned to formals as before, *except* that the matched tuple remains in tuple space. The "`rd("p",?x);`" operation in Fig. 3 has to wait for the termination of `p()` to read the return value of `p()`. It is presupposed that the return value of `p()` has the same type that the variable `x` is declared with.

`inp(template) / rdp(template)` In C-Linda, these operations attempt to locate a matching tuple and return `0` if they fail; otherwise, they return `1` and perform actual-to-formal assignment as described above. The only difference with `in/rd` is that the predicates will not block if no matching tuple is found.

A tuple and a template match in C-Linda, iff

- the tuple is passive, and
- the numbers of fields are equal, and
- types and values of actuals in templates are equal to the corresponding tuple fields, and
- the types of the variables in the formals are equal to the types of the corresponding tuple fields.

A *parallel programming language* consists of a coordination language like Linda and a sequential computation language. The first computation language, in which Linda has been integrated, is C. Meanwhile there exist also integrations into higher-level languages such as Smalltalk, Lisp, and Prolog. Implementations of C-Linda have been performed on a wide variety of parallel architectures: shared-memory multi-processors as well as on distributed memory architectures.

D The informal semantics of PROSET-Linda

This appendix provides a brief informal definition of generative communication in PROSET. For a more detailed discussion and some examples see [Hasselbring, 1991]. As PROSET is a still evolving language, some changes were made compared to the version in [Hasselbring, 1991]. The more interesting changes are mentioned in footnotes.

D.1 Process creation

In this section we will present an adaptation of the approach for process creation known from Multilisp to set-oriented programming, where new processes may be spawned inside and outside of tuple space. We regard tuple spaces primary as a device for synchronization and communication between processes, and only secondary for process creation.

D.1.1 Multilisp's futures

Multilisp [Halstead, 1985] augments Scheme with the notion of *futures* where the programmer needs no knowledge about the underlying process model, inter-process communication or synchronization to express parallelism. He only indicates that he does not need the result of a computation immediately (but only in the "future") and the rest is done by the runtime system. Instead of returning the result of the computation, a placeholder is returned as result of process spawning. The value for this placeholder is undefined until the computation has finished. Afterwards the value is set to the result of the parallel computation: the future *resolves* to the value. Any process that needs to know a future's value will be suspended until the future resolves thus allowing concurrency between the *computation* of a value and the *use* of that value. The programmer is responsible for ensuring that potentially concurrently executing processes in Multilisp do not affect each other via side effects. An example:

```
(let ((x (future expr1))
      (y expr2))
     ( body ))
```

The value for `x`, which will be the result value of *expr1*, is evaluated concurrently to *expr2* and *body*. The value for `y`, which will be the result value of *expr2*, is evaluated before the evaluation of *body* will be started. When *body* needs the value of `x`, and `x` is not yet resolved, it *touches* the future of `x` and is suspended until the future resolves. Most operations, e.g. arithmetic, comparison, type checking, etc., touch their operands. This is opposed to simple *transmission* of a value from one place to another, e.g. by assignment, passing as a parameter to a procedure, returning as a result from a procedure, building the value into a data structure, which does **not** touch the value. Transmission can be done without waiting for the value.

D.1.2 Process creation in PROSET

Futures in Multilisp provide a method for process creation but no means for synchronization and communication between processes, except for waiting for each other's termination. In our approach the concept for process creation via futures is adapted to set-oriented programming and combined with the concept for synchronization and communication using tuple spaces.

Multilisp is based on Scheme, which is a dialect of Lisp with lexical scoping. Lisp and Scheme manipulate pointers. This implies touching in a value-requiring context and transmission in a value-ignoring context. This is in contrast to PROSET that uses value semantics, i.e. a value is never transmitted by reference. However, there are a few cases where we can ignore the value of an expression: if the value of an expression is assigned to a variable, we do not need this value immediately, but possibly in the *future*.

Process creation in ProSet is provided through the unary operator `||`, which may be applied to an expression (preferably a function call). A new process will be spawned to compute the value of this expression concurrently with the spawning process analogously to futures in Multilisp. If this *process creator* `||` is applied to an expression that is immediately assigned to a variable, the spawning process continues execution without waiting for the termination of the newly spawned process. At any time the *value* of this variable is needed, the requesting process will be suspended until the future resolves (the corresponding process terminates) thus allowing concurrency between the *computation* and the *use* of a value. Consider the following statement sequence to see an example:

```
x := || p();        -- statement 1
...                 -- Some computations without access to x
y := x;             -- statement 2
```

After statement 1 is executed the process `p()` runs in parallel with the spawning process. Statement 2 will be suspended until `p()` terminates, because a copy is needed (value semantics). Also, if a compound data structure is constructed via a set or tuple forming enumeration, and this data structure is assigned immediately to a variable, we do not need the values of the enumerated components immediately, thus the following statement allows concurrency as above:

```
x := { || p(), 123, || q() };
```

If you replace statement 1 in the previously discussed statement sequence by this statement, then concurrency would be achieved as before. If the process creator `||` is applied in an expression that is an operand to any operator, then this operator will wait for the return value of the created process.

In summary: concurrency is achieved only at creation time of a process and maintained on immediately assigning to a variable, storing in a data structure, passing as a parameter to a procedure, returning as a result from a procedure, and depositing in tuple space (this is discussed in section D.2.1). Every time one tries to obtain a copy one has to wait for the termination of the corresponding process and obtains only then the returned value.

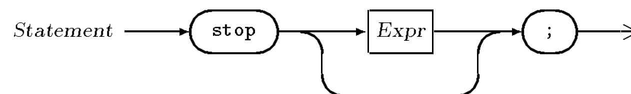
Also statements, which spawn new processes, are allowed:

```
|| p();
```

Side effects and write parameters are not allowed for processes. Communication and synchronization is done only via tuple-space operations. However, processes may access common, persistent data objects via files or *P*-files. This is discussed elsewhere [Doberkat *et al.*, 1992].

D.1.3 Program and process termination

The `stop` statement terminates the execution of a process or of an application program:

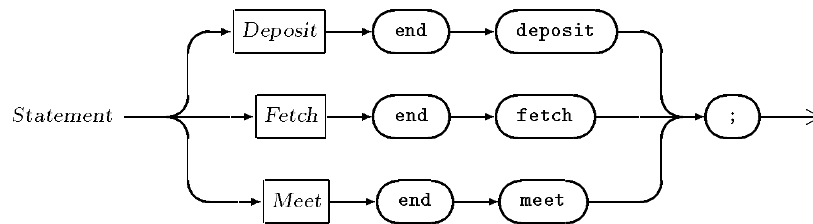


When executed in a spawned process, then this process will be terminated in the same way, as if a `return` statement with the same expression had been executed in the main procedure of this process. If no expression is specified, then `om` will be returned as usual.

When executed in a main program, which has been started form the operating system, then the value of the optional expression is passed to the operating system. Its meaning depends on the operating system. If no expression is specified, then `om` is passed to the operating system by default as success code. There exists implicitly a "`stop om;`" statement at the end of each main program. Termination of the main program terminates the entire application and thus all spawned processes.

D.2 Tuple-space operations

ProSet provides three tuple-space operations:

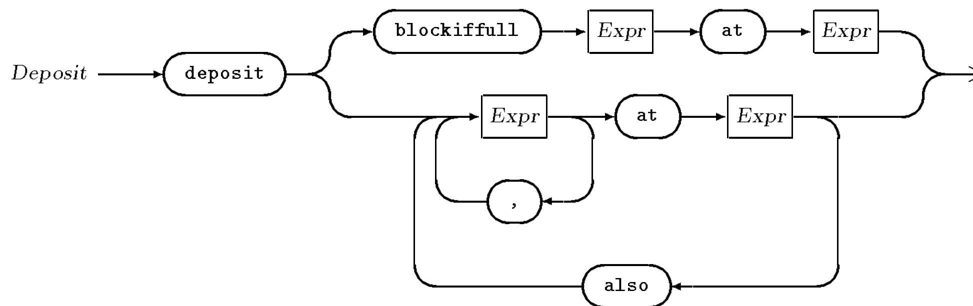


The **deposit** operation deposits new tuples into tuple space, the **fetch** operation fetches and removes a tuple from tuple space, and the **meet** operation meets and leaves a tuple in tuple space. It is possible to change the tuple's value while meeting it.

There is no difference between ProSet-tuples and passive Linda-tuples. Linda and ProSet both provide tuples thus it is quite natural to combine them on the basis of this common feature. Tuple-space operations are statements that yield **no** values. They should not be confused with operators in expressions that always yield values.

D.2.1 Depositing tuples

The **deposit** operation deposits tuples into a specified tuple space:



It is possible to deposit several tuples in an expression list into one tuple space and several such expression lists into multiple tuple spaces by one statement, but there are no guarantees made for the chronological order of availability of these tuples for other operations that wait for them (see below). The tuples are handed over to the tuple-space manager, which adds them to the tuple space in an arbitrary order.

All expressions are evaluated in arbitrary order, before any tuple is put into tuple space. The expressions must yield tuples to be deposited in tuple space; if not, the exception **type_mismatch** will be raised. Multiple tuple spaces will be discussed in section D.3.

We distinguish between passive and active tuples in tuple space. If there are no executing processes in a tuple, then this tuple is added as a passive one (cp. **out** of C-Linda). If there are executing processes in a tuple, then this tuple is added as an active one to tuple space. Depositing a tuple into tuple space does not touch the value. When all processes in an active tuple have terminated their execution, then this tuple converts into a passive one with the return values of these processes in the corresponding tuple fields. Active tuples are invisible to the other tuple-space operations until they convert into passive tuples. The other two tuple-space operations apply only to passive tuples.

Limited tuple spaces

Because every existing computing system has only finite memory, the memory for tuple spaces will also be limited. Pure tuple-space communication does not deal with *full* tuple spaces: there is always enough room available. Thus most runtime systems for Linda hide the fact of limited memory from the programmer. In ProSet-Linda the predefined exception `ts_is_full` will be raised by default when no memory is available for a `deposit` operation. If there are multiple tuples specified in one `deposit` operation, then none of them has been deposited when `ts_is_full` is raised. Conversely, this exception will be raised, if at least one tuple cannot be deposited in any tuple space.

`ts_is_full` is raised with the `signal` statement of ProSet. Such an exception can be resumed or aborted. If the handler then executes a `return` statement, the statement following the `deposit` will be executed and none of the tuples of the respective `deposit` will be deposited (abort). If the handler executes a `resume` statement, then the `deposit` operation tries again to deposit the tuples. See [Doberkat *et al.*, 1992] for a detailed discussion of exception handling in ProSet.

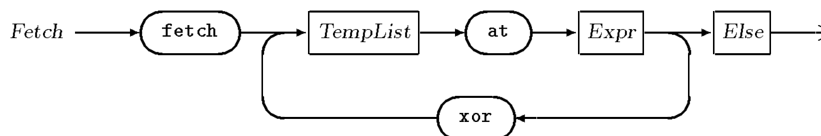
Optionally, the programmer may specify that a `deposit` operation will be suspended on a full tuple space until space is available again:

```
deposit blockiffull
        [ x ] at TS
end deposit;
```

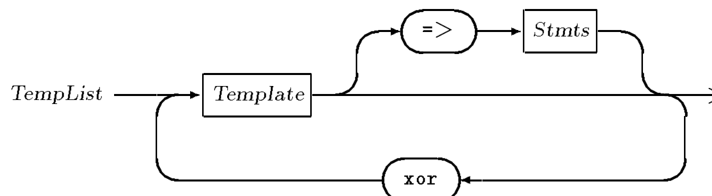
As you can see in the above syntax diagram, only one tuple may be deposited by a blocking `deposit` operation. If we would allow to block for multiple tuples, it would not be obvious if we should deposit them incrementally or all at once. Depositing all at once is in general not possible and incremental depositing may produce strange results if only a subset of the specified tuples can be deposited. Because of the unclear semantics, we do not allow multiple tuples for a blocking `deposit` operation. You can use multiple `deposit` operations instead.

D.2.2 Fetching tuples

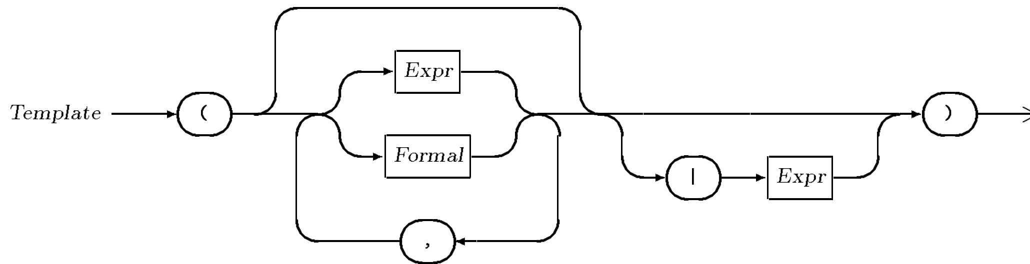
A `fetch` operation fetches and removes one tuple from a tuple space:



It is possible to specify several templates for multiple tuple spaces in one statement, but only one template may be selected nondeterministically (section D.2.4). We use the keyword `xor` (exclusive or) and not `or` to separate the individual alternatives, because only one template may be selected. If there are no `else` statements specified (see below) then the statement suspends until a match occurs. The selected tuple is removed from tuple space. If statements are specified for the selected template, these statements are executed (only for this template):



A template consists of a list of ordinary expressions and so-called *formals*:



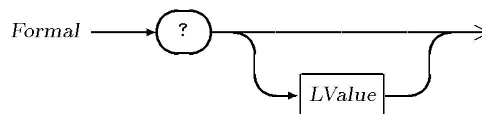
The expressions are called *actuals*. They are at first evaluated in arbitrary order. The list is enclosed in parentheses and not in brackets in order to set the templates apart from tuples. Note that a template may be empty to match the empty tuple `[]`.[3]

As usual | means *such that*. The Boolean expression behind | may be used to customize matching. ProSet employs *conditional value matching* and not the type matching known from C-Linda and similar embeddings of Linda into statically typed languages.

A tuple and a template match, iff all the following conditions hold:

- The tuple is passive.
- The arities are equal. Trailing `oms` do not contribute to the arity of tuples and templates.[4]
- Values of actuals in templates are equal to the corresponding tuple fields.
- The Boolean expression behind | in the template evaluates to `true`. If no such expression is specified, then `|true` is the default.

The fields that are preceded by a question mark are the *formals* of the template:



The *l*-values specified in the formals are assigned the values of the corresponding tuple fields provided matching succeeds. If an *l*-value is specified more than once, it is not determined which of the possible values is assigned. If no *l*-value is specified, then the corresponding value will not be available. You may regard a formal without an *l*-value as a "don't care" or "only take care of the condition" field. The symbol `$` may be used like an expression as a placeholder for the value of the corresponding tuple in tuple space.[5] The condition is enclosed within the parentheses to make the implicit scope of `$` explicit.

[3] In [Hasselbring, 1991] empty template lists were not allowed, whereas empty tuples may be deposited in tuple space. This inconsistency was derived form the heritage of C-Linda, where tuples and templates must have at least one field. This is an example for a situation in which inconsistencies in an informal specification were found while constructing a formal specification.

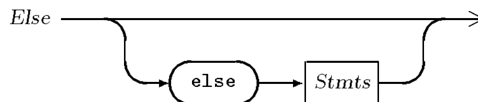
[4] In [Hasselbring, 1991] we did not specify how to handle templates with trailing `oms` while matching. This is an example for a situation in which an omission in an informal specification was found while constructing a formal specification.

[5] In [Hasselbring, 1991] conditions were specified at individual tuple fields, where the symbol `$` was used as a placeholder for the value of the corresponding tuple field and not for the entire tuple. This implied that it was not possible to access the values of multiple tuple fields within such expressions. Now we have a more flexible and concise way for specifying conditions for matching.

Note that the template `(1,om)` matches the tuple `[1]`, and that the template `(om,1)` does not match the tuple `[1]`. If we would assign the arity `2` to the template `(1,om)` then no tuple could ever match this template.

Non-blocking matching

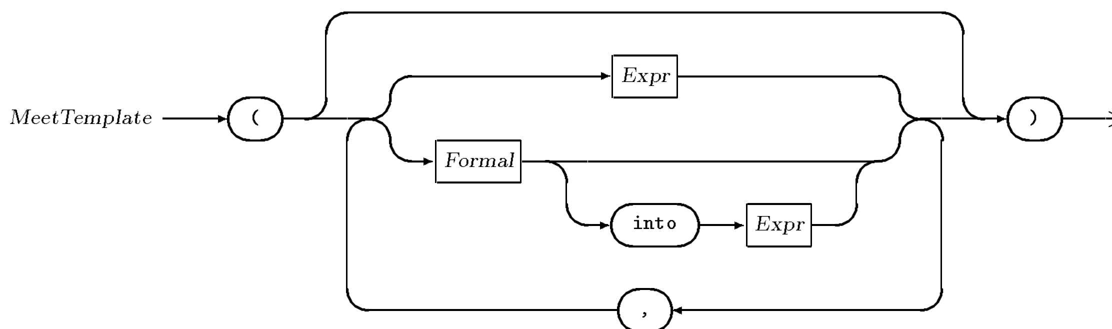
It is possible to specify `else` statements to be executed, if none of the templates matches:



We will use the notion *non-blocking matching* if `else` statements are specified as opposed to *blocking matching* if no `else` statements are specified.

D.2.3 Meeting tuples

The `meet` operation meets and leaves one tuple in tuple space. It is possible to change the tuple while meeting it. Exchanging the keyword `fetch` with `meet` and the nonterminal *Template* with *MeetTemplate* in the first syntax diagrams of section D.2.2, one obtains the syntax for the `meet` operation:



The expressions are evaluated as usual, the formals are used to create templates, which are used for matching as with the `fetch` operation (section D.2.4). If no `else` case is specified, then the statement suspends until a match occurs. The values of the tuple fields that were fetched for the corresponding formals of the template are assigned to the corresponding *l*-values. If statements are specified for the selected template, these statements are executed (only for this template).

If there are no `into`s specified behind the formals, then the selected tuple is **not** removed from tuple space. This case may be compared with the `rd/rdp` operations of C-Linda. Except for the fact that the `meet` operation without `into`s leaves the tuple it found in tuple space, it works like the `fetch` operation.

Changing tuples

We allow to change tuples while meeting them in tuple space. This is done by specifying expressions `into` which specific tuple fields will be changed. Tuples, which are met in tuple space, may be regarded as shared data since they remain in tuple space; irrespective of changing them or not.

If there are `into`s specified behind the formals then the tuple is at first fetched from the tuple space as it would be done with the `fetch` operation. Afterwards a tuple will be deposited into the same tuple space, where all the tuple fields without `into`s are unchanged and all the tuple fields with `into`s are updated with the values of the respective expressions.

D.2.4 Nondeterminism and fairness while matching

There are two sources for nondeterminism while matching:

1. Several matching tuples exist for the templates: one tuple will be selected nondeterministically.
2. The selected tuple matches several templates: one template will be selected nondeterministically.

If in any case there is only one candidate available, this one will be selected. There are several ways for handling fairness while selecting tuples or templates that match if there are multiple candidates available. We assume a fair scheduler to guarantee process fairness, which means that no single process is excluded of CPU time forever. We will now discuss *fairness of choice* which is important for handling the nondeterminism derived from matching. There exist some fairness notions:

Weak Fairness If a process is enabled continuously from some point onwards then it eventually will be selected. Weak Fairness is also called *justice*.

Strong Fairness If a process is enabled infinitely often then it will be selected infinitely often.

In ProSet the following fairness guarantees are given for the two sources for nondeterminism as mentioned above:

1. Tuples will be selected without any consideration of fairness.
2. Templates will be selected in a weakly fair way.

Since deposited tuples are no longer connected with processes, it is reasonable to select them without any consideration of fairness. Linda's semantics do not guarantee tuple ordering — this aspect remains the responsibility of the programmer. If a specific order in selection is necessary, it has to be enforced via appropriate tuple contents. The system is enabled to store the tuple space e.g. in a hash-based way.

Fairness is also important for processes which are blocked on full tuple spaces:[6]

3. Processes which are blocked on full tuple spaces are selected in a weakly fair way when tuples are fetched from the respective tuple spaces.

In cases 2 and 3 processes are involved and enabled after selection, whereas in case 1 this is not the case for the selection of deposited tuples. Therefore, it is reasonable to employ weakly fair selection in cases 2 and 3, and unfair selection in case 1.

To specify weakly fair selection of templates by means of temporal logic we introduce the following abbreviations for predicates on a process P and a template T:

[6] In [Hasselbring, 1991] we did not specify how to select processes which are blocked on full tuple spaces. This is another example for a situation in which an omission in an informal specification was found while constructing a formal specification.

$$
\begin{aligned}
\mathbf{E} &= \text{“Process } P \text{ \underline{e}xecutes a blocking \texttt{fetch} or \texttt{meet} operation with template } T\text{”} \\
\mathbf{M} &= \text{“There is a \underline{m}atching tuple for template } T \text{ in tuple space”} \\
\mathbf{B} &= \text{“Process } P \text{ is \underline{b}locked with template } T\text{”} \\
\mathbf{S} &= \text{“Template } T \text{ is \underline{s}elected for a matching tuple provided there exists one”} \\
\mathbf{A} &= \text{“Process } P \text{ is \underline{a}ctivated (template } T \text{ was selected)”}
\end{aligned}
$$

Now the above-given fairness guarantee may be formulated as follows:[7]

$$\mathbf{E} \wedge \mathbf{M} \wedge \mathbf{S} \Rightarrow \mathbf{A} \tag{1}$$
$$\mathbf{E} \wedge \neg(\mathbf{M} \wedge \mathbf{S}) \Rightarrow \mathbf{B} \tag{2}$$
$$\mathbf{B} \wedge \Box\Diamond\mathbf{M} \Rightarrow \Diamond(\mathbf{S} \wedge \mathbf{A}) \tag{3}$$

Predicates 1 and 2 describe the behavior on executing blocking `fetch` or `meet` operations. Predicate 3 describes the selection of a template that belongs to a suspended process. “$\Box\Diamond\mathbf{M}$” means that there is infinitely often a matching tuple available for the template. **S** (selection) is implied by “$\Box\Diamond\mathbf{M}$”.

This is not a formal specification, since the predicates are based on English text and not on mathematical formulas. In section 2.8 a formal specification is given.

Weakly fair selection of templates applies only to blocking matching: if a template that is used for non-blocking matching does match immediately then this one is excluded of further matching and the corresponding process is informed of this fact. This applies accordingly to non-blocking matching with multiple templates, too. Templates (resp. processes), which are suspended because no tuple matches them are weakly fair matched with tuples later deposited. The implementation has to guarantee this.

D.3 Multiple tuple spaces

Atoms are used to identify tuple spaces. As mentioned in appendix A atoms are unique for one machine and across machines. They have first-class rights.

The expressions behind the keyword `at` within tuple-space operations have to yield valid tuple-space identities. If not, the exception `ts_invalid_id` will be raised. Note that `ts_invalid_id` and not `ts_is_full` will be raised if both exceptional conditions — invalid tuple-space identity and full tuple space — hold. Note also that the exception `type_mismatch` and not `ts_invalid_id` will be raised if an expression behind the keyword `at` within tuple-space operations yields *not* an atom.

ProSet provides several library functions to handle multiple tuple spaces:

`CreateTS(limit):` Calls the standard function `newat` to return a fresh atom. The tuple-space manager is informed to create a new tuple space represented/identified by this atom. The atom will be returned by `CreateTS`. Therefore, you can only use atoms that were created by `CreateTS` to identify tuple spaces.

The integer parameter `limit` specifies a limit on the expected or desired size of the new tuple space. This size limit denotes the total number of passive and active tuples, which are allowed in a tuple space at the same time. `CreateTS(om)` would instead indicate that the expected or wanted size is unlimited regarding user-defined limits, not regarding physical limits. A negative limit is equivalent to 0 (no tuples may be deposited into such a tuple space).

`ExistsTS(TS):` Yields `true`, if `TS` is an atom that identifies an existing tuple space; else `false`.

[7] In temporal logic “$\Diamond p$” and “$\Box p$” mean that predicate p holds eventually resp. always. See e.g. [Emerson, 1990] for a full account to temporal logic.

`ClearTS(TS)`: Removes all active and passive tuples from the specified tuple space.

`RemoveTS(TS)`: Calls `ClearTS(TS)` and removes `TS` from the list of existing tuple spaces.

If these functions are invoked with actual parameters that are not atoms, the exception `type_mismatch` will be raised. If the functions `ExistsTS`, `ClearTS`, or `RemoveTS` are called with an atom, which is not a valid tuple-space identity, then the exception `ts_invalid_id` will be raised.

Every ProSet program has its own tuple-space manager. Tuple spaces are not persistent. They exist only until all processes of an application have terminated their execution.

E Types of all names defined globally

This appendix has been produced by the *fuzz* type-checker for Z with the `-t` flag [Spivey, 1992a]. A comprehensive overview of all names defined globally in the specification with their associated types is given. Some line breaks were inserted to fit into two columns. An index to these global names may be found in the index of formal definitions at the end of this document.

```
Given Expression

Given LValue

Given Statement

Given Process

Var Execute _: P Statement

Given Value

Var atom: Value

Var boolean: Value

Var integer: Value

Var real: Value

Var string: Value

Var tuple: Value

Var set: Value

Var function: Value

Var modtype: Value

Var instance: Value

Var TRUE: Value

Var FALSE: Value

Var om: Value

Var ValuesOfType: Value -+> P Value

Var Type: Value -+> Value

Var Evaluate: Expression -+> Value

Var ProcRetVal: Process -+> Value

Var _ \IsAssigned _: LValue <-> Value

Given TupleComp

Var TupleValue: Value -+> TupleComp

Var TupleProcess: Process -+> TupleComp

Abbrev APTuple: P (seq TupleComp)

Var APTupleToValue: APTuple -+> Value

Genconst Arity[1]:
    (Value -+> @1) -+> (seq @1 -+> NN)

Var TupArity: APTuple -+> NN

Given OptLValue

Var NoLValue: OptLValue

Var IsLValue: LValue -+> OptLValue

Given OptInto

Var NoInto: OptInto

Var IsInto: Expression -+> OptInto

Schema Formal
    Destination: OptLValue
    Into: OptInto
End

Given TempComp

Var TempValue: Value -+> TempComp

Var TempFormal: Formal -+> TempComp

Schema Template
    List: seq TempComp
    Condition: Expression
End

Var _ \FormalAssign _: Template <-> APTuple

Var _ \EvalIntos _:
             Template x APTuple -+> APTuple

Var TempArity: seq TempComp -+> NN

Var _ \CompMatches _: TupleComp <-> TempComp

Var _ \Matches _: APTuple <-> Template

Given OptStmt
```

```
Var NoStmt: OptStmt

Var IsStmt: Statement -+> OptStmt

Var OptExecute _: P OptStmt

Schema Pending
    temp: Template
    os: OptStmt
    proc: Process
End

Schema TupleSpace
    Id: Value
    Limit: Value
    Tuples: bag APTuple
    PendFetch: F Pending
    PendMeet: F Pending
    PendFull: Process -++> APTuple
End

Schema Program
    TSs: F TupleSpace
    ActiveProcs: F Process
End

Var ActuallyActiveProcesses: F Process

Var ActuallyPendingProcesses: F Process

Var ActuallyExistingProcesses: F Process

Schema \Delta Program
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Schema InitProgram
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Schema ProcessCreation
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    NewProcess?: Process
End

Var _ \SAR _:
    F Process x F Pending -+> F Pending

Schema ProcessTermination
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    ToKill?: Process
End

Schema ProgramTermination
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Var IDsOF: F TupleSpace -+> F Value

Schema CreateTSok
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InLimit?: Value
    Return!: Value
End

Abbrev \TypeMismatch: Statement

Schema \Xi Program
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Schema CreateTSTypeMismatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InLimit?: Value
    Exception!: Statement
End

Schema CreateTS
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InLimit?: Value
    Return!: Value
    Exception!: Statement
End

Schema ExistsTS
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Return!: Value
```

```
    Exception!: Statement
End

Schema ClearTSok
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Return!: Value
End

Abbrev \InvalidId: Statement

Schema ClearTSinvalid
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Exception!: Statement
End

Schema ClearTS
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Return!: Value
    Exception!: Statement
End

Schema RemoveTSfromState
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
End

Schema RemoveTS
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTS?: Value
    Return!: Value
    Exception!: Statement
End

Abbrev TupList: P (seq (seq APTuple x Value))

Var HasIntos _: P Template

Var _ \AllTuples _:
            Value x TupList -+> seq APTuple

Genconst BagSum[1]: bag @1 -+> NN

Var IntValueOf: Value -+> ZZ

Schema TSisFull
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTupList?: TupList
End

Var _ \AddTuple _:
    Program x (APTuple x Value) -+> Program

Schema DepositOK
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTupList?: TupList
End

Schema DepositInvalid
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTupList?: TupList
    Exception!: Statement
End

Given BlockMode

Var BlockIfFull: BlockMode

Var DoNotBlock: BlockMode

Abbrev \ExcTSisFull: Statement

Schema FullTSException
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTupList?: TupList
    Blocking?: BlockMode
    Exception!: Statement
End

Schema FullTSBlock
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTupList?: TupList
    Blocking?: BlockMode
    InProc?: Process
End
```

```
Schema Deposit
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InTupList?: TupList
    Blocking?: BlockMode
    InProc?: Process
    Exception!: Statement
End

Abbrev TempList:
  P (seq (seq (Template x OptStmt) x Value))

Genconst GetTS[4]: @1 x @2 x @3 x @4 -+> @1

Genconst GetTemp[4]: @1 x @2 x @3 x @4 -+> @2

Genconst GetTup[4]: @1 x @2 x @3 x @4 -+> @3

Genconst GetOS[4]: @1 x @2 x @3 x @4 -+> @4

Schema FetchMatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
End

Schema FetchNoMatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
End

Schema DoElseStmt
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
End

Schema InvalidTempList
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Exception!: Statement
End

Schema DisallowIntos
    InTempList?: TempList
End

Schema Fetch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
    Exception!: Statement
End

Schema MeetMatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
End

Schema MeetNoMatch
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
End

Schema Meet
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
    InProc?: Process
    InTempList?: TempList
    Else?: OptStmt
    Exception!: Statement
End

Schema NoOp
    TSs: F TupleSpace
    ActiveProcs: F Process
    TSs': F TupleSpace
    ActiveProcs': F Process
End

Schema Execution
    History: NN -+> \Delta Program
End

Schema TupleSpaceDesign
```

```
    Id: Value
    Limit: Value
    Tuples: bag APTuple
    PendFetchDesign: seq Pending
    PendMeetDesign: seq Pending
    Blocked: Process -++> APTuple
End

Schema Abstraction
    Id: Value
    Limit: Value
    Tuples: bag APTuple
    PendFetch: F Pending
    PendMeet: F Pending
    PendFull: Process -++> APTuple
    PendFetchDesign: seq Pending
    PendMeetDesign: seq Pending
    Blocked: Process -++> APTuple
End
```

References

[Anderson *et al.*, 1990] H. Anderson, P.D. Fabricius, and M.G. Jensen. Linda & Logic. Master's thesis, University of Aalborg, Dept. Computer Science, Denmark, June 1990.

[Bosschere and Wulteputte, 1991] K. De Bosschere and L. Wulteputte. Multi-Prolog: Implementation on an 88000 Shared Memory Multiprocessor. Technical Report DG 91-19, University of Gent, LEM, B-9000 Gent, December 1991.

[Butcher, 1991] P. Butcher. A behavioural semantics for Linda-2. *IEE/BCS Software Engineering Journal*, 6(4):196–204, July 1991.

[Callsen *et al.*, 1991] C.J. Callsen, I. Cheng, and P.L. Hagen. Optimizing Linda. Master's thesis, University of Aalborg, Dept. Computer Science, Denmark, June 1991.

[Carriero and Gelernter, 1990] N. Carriero and D. Gelernter. *How to write parallel programs.* MIT Press, 1990.

[Ciancarini *et al.*, 1992] P. Ciancarini, K.K. Jensen, and D. Yanklevich. The semantics of a parallel language based on a shared dataspace. Technical Report 26/92, University of Pisa, July 1992.

[DeMillo *et al.*, 1979] R.A. DeMillo, R.J. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979.

[Dijkstra, 1988] E.W. Dijkstra. Position paper on "fairness". *ACM SIGSOFT Software Engineering Notes*, 13(2):18–20, April 1988.

[Diller, 1990] A. Diller. *Z: An introduction to formal methods.* John Wiley and Sons, 1990.

[Doberkat *et al.*, 1992] E.-E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — Prototyping with Sets: Language Definition. Informatik-Bericht 02-92, University of Essen, April 1992.

[Emerson, 1990] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier, 1990.

[Gelernter, 1985] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Halstead, 1985] R.H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Hasselbring, 1991] W. Hasselbring. On Integrating Generative Communication into the Prototyping Language ProSet. Informatik-Bericht 05-91, University of Essen, December 1991.

[Hazelhurst, 1990] S. Hazelhurst. A proposal for the formal specification of the semantics of Linda. Technical Report 1990-14, Department of Computer Science, University of the Witwatersrand, Johannesburg, South Africa, October 1990.

[Jagannathan, 1990] S. Jagannathan. Semantics and analysis of first-class tuple spaces. Research Report 783, Yale University, New Haven, CT, April 1990.

[Jensen, 1990] K.K. Jensen. The semantics of tuple space and correctness of an implementation. Research Report 788, Yale University, New Haven, CT, April 1990.

[Jensen, 1992] K.K. Jensen. *Towards a Multiple Tuple Space Model.* PhD thesis, University of Aalborg, Dept. Computer Science, Denmark, 1992. (in preparation).

[Leichter, 1989] J.S. Leichter. *Shared tuple memories, buses and LAN's — Linda implementations across the spectrum of connectivity.* PhD thesis, Yale University, New Haven, CT, July 1989.

[Meira and Cavalcanti, 1992] S.R.L. Meira and A.L.C. Cavalcanti. The MooZ Specification Language. Relatório Ténico ES/1.92, University of Pernambuco, Recife, Brasil, January 1992.

[Milner, 1989] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Narem, 1989] J.E. Narem. An informal operational semantics of C-Linda V2.3.5. Technical Report 839, Yale University, New Haven, CT, December 1989.

[Parker, 1991] C.E. Parker. Z tools catalogue. Technical Report ZIP/BAe/90/020, British Aerospace, Warton, UK, May 1991.

[Spivey, 1992a] J.M. Spivey. *The fUZZ Manual.* Computing Science Consultancy, Oxford, UK, 2nd edition, July 1992.

[Spivey, 1992b] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2nd edition, 1992.

Index of formal definitions

⊞ 10

A

Abstraction 31
ActuallyActiveProcesses 10
ActuallyExistingProcesses 10
ActuallyPendingProcesses 10
AddTuple 18
AllTuples 17
APTuple 5
APTupleToValue 5
Arity 5
atom 4

B

BagSum 17
BlockIfFull 21
BlockMode 21
boolean 4

C

ClearTS 15
ClearTSinvalid 14
ClearTSok 14
CompMatches 7
CreateTS 13
CreateTSok 12
CreateTSTypeMismatch 12

D

Deposit 22
DepositInvalid 21
DepositOK 21
DisallowIntos 25
DoElseStmt 24
DoNotBlock 21

E

'`escape ts_invalid_id();`' 14
'`escape type_mismatch();`' 12
EvalIntos 6
Evaluate 4
Execute 3
Execution 27
ExistsTS 13
Expression 3

F

FALSE 4
Fetch 25
FetchMatch 23
FetchNoMatch 24
Formal 6
FormalAssign 6
FullTSBlock 21
FullTSException 21
function 4

G

GetOS 22
GetTemp 22
GetTS 22
GetTup 22

H

HasIntos 17

I

IDsOF 12
InitProgram 10
instance 4
integer 4
IntValueOf 18
InvalidTempList 25
IsAssigned 4
IsInto 6
IsLValue 6
IsStmt 8

L

LValue 3

M

Matches 7
Meet 27
MeetMatch 26
MeetNoMatch 27
modtype 4

N

NoInto 6
NoLValue 6
NoOp 27
NoStmt 8

O

om 4
OptExecute 8
OptInto 6
OptLValue 6
OptStmt 8

P

Pending 8
Process 3
ProcessCreation 10
ProcessTermination 11
ProcRetVal 4
Program 9
ProgramTermination 11

R

real 4
RemoveTS 15
RemoveTSfromState 15

S
set 4
'`signal ts_is_full();`' 21
Statement 3
string 4

T
TempArity 7
TempComp 6
TempFormal 6
Template 6
TempList 22
TempValue 6
TRUE 4
TSisFull 18
TupArity 6
tuple 4
TupleComp 5
TupleProcess 5
TupleSpace 8
TupleSpaceDesign 31
TupleValue 5
TupList 17
Type 4

V
Value 3
ValuesOfType 4

Index of explained standard Z symbols and keywords

_ 3
⩁ 11
$\bullet$ 40
θ 38
$\subset$ 4
$\subseteq$ 4
$\mapsto$ 4
Ξ 38
$\uplus$ 9
$\sharp$ 9
$\frown$ 17
$'$ 19, 38
$\longleftrightarrow$ 4
$\gg$ 15
. . 5
| 39, 40
\ 5
Δ 38
$^{-1}$ 6
$\lhd$ 11
$\times$ 7
⨾ 15
$\wedge$ 13
$\vee$ 13, 38
$\widehat{=}$ 13, 38
! 10, 38
. 6
: 37
::= 39
= 4
== 37
? 10, 38

{} 39, 40
[] 37, 38
⟦⟧ 11
⟨⟩ 39
⟪⟫ 39
$\longrightarrow$ 4
⇸ 4
⇻ 9
↣ 39
⤔ 4
⤖ 37

bag 8

dom 4

$\exists_1$ 12

$\mathbb{F}$ 9

false 4
first 10
front 17

head 17

iseq 31

last 17

max 5

$\mathbb{N}$ 37
$\mathbb{N}_1$ 5

$\mathbb{P}$ 3
partition 39

ran 39

second 10
seq 5
seq_1 17

tail 17
true 4

$\mathbb{Z}$ 37