Report Number: WUCS-TM-92-03

1992-11-01

Swarm Language Reference Manual

Jerome Y. Plun, C. Donald Wilcox, and Kenneth C. Cox

This document contains a description of the grammar and syntax rules of the Swarm programming language. This language, which is an implementation of the program specification language used with the Swarm computational model, is used both to specify programs and their visualizations, for the use of the Pavane program visualization system.

School of Engineering & Applied Science

Swarm Language Reference Manual

Jerome Y. Plun
C. Don Wilcox
Kenneth C. Cox

WUCS-TM-92-03

November 1992

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899, USA

Abstract

This document contains a description of the grammar and syntax rules of the Swarm programming language. This language, which is an implementation of the program specification language used with the Swarm computational model, is used both to specify programs and their visualizations, for use with the Pavane program visualization system.

Correspondence: All communications regarding this report should be addressed to:

Dr. Gruia-Catalin Roman
Department of Computer Science
Washington Universty
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899, USA

office: (314) 935 6190
secretary: (314) 935 6160
fax: (314) 935 7302

roman@swarm.WUSTL.edu

Table of Content

List of Tables and Figures

1 Introduction

This document contains a description of the grammar and syntax rules of the Swarm programming language. This language, which is an implementation of the program specification language used with the Swarm computational model, is used both to specify programs and their visualizations, for use with the Pavane program visualization system.

2 Lexical Conventions

The following section describes the basic elements, or tokens, of the Swarm programming language. A program is a sequence of tokens which satisfy the syntactic definition of the language as described in the later sections.

2.1 Comments

There are two comment delimiters in Swarm. The characters `/*` introduce a comment that terminates with the characters `*/`. They do not indicate a comment when occurring within a string literal. Comments can be nested. Once the `/*` introducing a comment is seen, all other characters are ignored until the matching `*/` is encountered.

Single-line comments are indicated by a double slash `//`. Everything on the program line to the right of the delimiter is treated as a comment and ignored by the compiler.

A comment is treated as a single whitespace character.

2.2 Identifiers

An identifier is a sequence of letters, digits, and underscores (_), possibly followed by one or more prime characters ('). The first character cannot be a digit. Uppercase and lowercase letters are distinct. Legal letters include both the standard roman and greek letters. Identifier length is limited to 32 characters.

2.3 Keywords

Because of the rich visualization language supported by the compiler, a large number of identifiers are reserved and cannot be used for any other purpose in a Swarm program. In an effort to reduce the number of conflicts caused by the large keyword set, reserved identifiers which are used as visualization and window attributes have a preceding underscore. To avoid possible conflicts in the future, progrmmers hsould avoid defining identifiers which begin with an underscore. The list of reserved keywords is shown in Figure 1, "Reserved keywords," on page 2.

2.4 Constants

The seven types of constants are *integer*, *real*, *string*, *set*, *list*, *array*, and *struct*.

An *integer* constant consists of a sequence of digits. A unary operator, such as "–", preceding the constant is not considered part of the constant.

A *real*, or floating point, constant consists of an integer part, a decimal point, and a fraction part. The integer and fraction parts are each a sequence of digits. Either the integer part or the fraction part (but not both) may be missing. As for an integer constant, a preceding unary operator is not considered part of a real constant.

A *string* constant is a sequence of characters surrounded by double quotes, as in `"..."`. A double-quote character (") inside a string constant must be preceded by a backslash (\). In addition, several escape sequences are used to represent some special characters, as specified in Table 2, "Escape Sequences," on page 3. These are the same escape sequences accepted by ANSI-compliant C compilers.

A *set* constant is a (possibly empty) sequence of expressions, all having the same data type, separated by commas , and surrounded by braces `{ }`.

AND	OR	_WinCenterZ
animation	_origin	_WinColorBlue
ARRAY	_point	_WinColorChange
array	_polygon	_WinColorGreen
_arrowfrom	_position	_WinColorRed
_arrowsize	program	_WinDepth
_arrowto	_radius	_WinDepthChange
boolean	_ramp	_WinDepthExp
_box	real	_WinDepthMax
_center	_rectangle	_WinDepthMin
_chop	rules	_WinDistance
_circle	SET	_WinDistanceChange
_color	set	_WinDistanceExp
constant	_shapecylinder	_WinDistanceMax
_corner	skip	_WinDistanceMin
_crect	space	_WinIncidence
definitions	_sphere	_WinIncidenceChange
_direction	_square	_WinIncidenceSpin
double	state	_WinIncidenceSpinChange
end	static	_WinPositionChange
enum	_step	_WinPositionX
_facefill	string	_WinPositionXMax
_facefillcolor	struct	_WinPositionXMin
false	_T_	_WinPositionY
_fill	tail	_WinPositionYMax
_fillcolor	_text	_WinPositionYMin
float	then	_WinRoll
_from	_to	_WinRollChange
functions	transaction	_WinRollSpin
head	TRUE	_WinRollSpinChange
include	true	_WinSizeChange
inf	tuple	_WinSizeX
initialization	types	_WinSizeXMax
integer	variable	_WinSizeXMin
_label	_vector	_WinSizeY
_length	_vertices	_WinSizeYMax
_lifetime	visualization	_WinSizeYMin
_line	_width	_window
list	_WinAzimuth	windows
max	_WinAzimuthChange	_xrad
min	_WinAzimuthSpin	_xrot
mod	_WinAzimuthSpinChange	_xsize
multiset	_WinBorder	_yrad
NAND	_WinCenterChange	_yrot
NOR	_WinCenterExp	_ysize
_octahedron	_WinCenterScrollExp	_zrot
of	_WinCenterX	_zsize
old	_WinCenterY	

Table 1: Reserved keywords

Character Name	Escape Sequence
new line	`\n`
horizontal tab	`\t`
vertical tab	`\v`
backspace	`\b`
carriage return	`\r`
form feed	`\f`
backslash	`\\`
single quote	`\'`
double quote	`\"`
question mark	`\?`

Table 2: Escape Sequences

A *list* or an *array* constant is similar to a set except that the sequence of expressions is surrounded by brackets `[ ]`.

A *struct* constant is a sequence of two or more expressions of possibly different data types, separated by commas (,), and surrounded by backets `[ ]`. The data type of an expression in square brackets is determined by its usage.

The elements of a set, list, array, or struct constant can be any constant type, including other sets, lists, arrays, or structs. For example, the following constant is perfectly valid.

```
[
   [ { 1, 2 }, { }, { 3 } ],
   { { }, { "a String", "another String" },
   { [ { 4.0 }, { 5.0 } ], [ ] },
   100,
   "that's it!"
]
```

2.5 Operators

An operator specifies an operation to be performed. The operators `[ ]`, `{ }`, `( )`, `| |`, and `?  :` must occur in pairs, possibly separated by expressions.

```
[ ] { } ( ) | | . ¬ †
~ ≈ + - * / mod ∪ ∩ ∈ ∉ ⊂ ⊆ ⊃ ⊇ ⊄ ∧ ∨
< > ≤ ≥ = ≠
? :
, :=
```

Table 3: Swarm operators

Particular operations are discussed in Section 7, "Expressions."

2.6 Punctuation Symbols

Punctuation symbols are tokens which have semantic significance without specifying an operation to be performed. The symbols ⟨ ⟩ and `( )` must occurs in pairs. Some operators are also punctuation symbols as determined by context.

; ⇐ → . \ ⟨ ⟩ ( )

Table 4: Swarm punctuation symbols

2.7 Multiglyph Forms

Several of the punctuation symbols and operators are not available in the 128-character ASCII character set. Table 2.7 describes the alternate, or multiglyph, form for these symbols. Note that it is not possible to use either the greek character set or the prime character in identifiers without using the 256 element Swarm character set. For the full Swarm character set, refer to appendix A A program is available for converting programs to and from the Swarm character set..

Swarm Glyph	Multiglyph	Swarm Glyph	Multiglyph	Swarm Glyph	Multiglyph
≤	<=	⇒	=>	Σ	\+
≥	>=	→	->	Π	*
≠	<>	⟨	<<	∀	\A
∧	/\	⟩	>>	∃	\E
∨	\/	⊂	.<	‖	||
¬	!	⊆	.<=	≈	~~
Ⓣ	head	∈	.?	≡	==
↬	tail	⇒	{}	†	@

Table 5: Multiglyph forms

3 Program Structure

Swarm and visualization programs are composed of a program header, a sequence of sections specifying various components of the program, and the keyword end. The different sections allowed in a Swarm program and a visualization program are shown in Figure 1, "Sections of a Swarm program and visualization," on page 4. Sections can occur multiple times, and the order is irrelevant, as long as any identifier is declared before it is used.

Swarm:	**Visualization**
Data Types	Data Types
External Functions	External Functions
Definition	Definition
Tuples Declaration	Spaces
Transactions Declaration	Windows
Transactions Definition	Rule Definition
Initialization	Initialization

Figure 1: Sections of a Swarm program and visualization

The program header specifies the name of the program and a (possibly empty) set of parameters provided to the computation. Each parameter is described by its name and its type (cf Section 4 for Data Types). The parameters are separated by commas. The formal definition of the Program Structure is shown in Figure 2, "BNF definition of the Program Structure," on page 5.

A Swarm program has a header starting with the keyword program while a visualization program starts with the

AProgram ::= `program` *ProgramHeader Sections* `end`
| `visualization` *ProgramHeader PipeDefinition Sections* `end`

ProgramHeader ::= `<Identifier>` (*ProgParmList*)

ProgParmList ::= *AType* `<Identifier>` | *ProgParmList* , *AType* `<Identifier>` | λ

PipeDefinition ::= `state` ⇒ *SpaceConnection* `animation` ;

SpaceConnection ::= | `<Identifier>` ⇒ *SpaceConnection*

Sections ::= *SectionsList* | λ

SectionsList ::= *ASection* | *SectionList ASection*

ASection ::=	*TypeSection*	(See "Data Types" on page 5)
	| *FunctionDeclaration*	(See "Functions" on page 7)
	| *DefinitionSection*	(See "Definitions" on page 9)
	| *SwarmTuples*	(See "Tuples and Transactions Declarations" on page 16)
	| *SwarmTransactions*	(See "Tuples and Transactions Declarations" on page 16)
	| *TransationSection*	(See "Transaction Definitions" on page 16)
	| *InitializationSection*	(See "Swarm Initialization" on page 17 or "Visualization Initialization" on page 20)
	| *ASpaceDeclaration*	(See "Visualization Spaces" on page 17)
	| *WindowsSection*	(See "Windows" on page 18)
	| *RuleDeclaration*	(See "Visualization Rules" on page 19)

Figure 2: BNF definition of the Program Structure

keyword `visualization`. `state` represents the input dataspace being visualized, `animation` the rendering space, and, in the rule for *SpaceConnection*, each `<Identifier>` token is an intermediary space used to map `state` into `animation [1]`.

The other sections are described in the rest of this document. The section number of each program section is indicated in the BNF definition of *ASection*. We first cover the 3 common sections (data types, external functions, and definitions), then complete the specification of a Swarm program (tuples and transactions declarations, transactions definitions, and initialization), and finally describe the remainder of a visualization (spaces, windows, rules, initialization).

Appendices B, C, and D contain an example of a Swarm program, a visualization for it, and the definition of the external functions used in this program and its visualization.

4 Data Types

The Swarm language supports four fundamental types of objects: *boolean*, *integer*, *real*, and *string*. More complex types are referred to as *aggregate* types.

4.1 Fundamental Types

The types *float* and *double* are synonyms for the *real* data type. Integer corresponds to a *long* in the C programming language, and *real* to a *double*.

Typical limits for an 32 bits long are -2,147,483,648 and 2,147,483,647. Typical limits for a 64 bit double are -1.7980E+308 and 1.7980E+308, with a precision of up to 2.225E-308.

The boolean data type has only the two values `true` and `false`.

4.2 Derived Types

Swarm provides 4 aggregate data types which can be used to construct a conceptually infinite class of derived data types. The aggregate data types are shown in Figure 3, "Aggregate Data Types," on page 6.

- *set* An unbounded and unordered group of elements of one specific type. A set of elements of type `aType` is specified as
  ```
  set of aType
  ```
  for example
  ```
  set of integer
  ```

- *list* An ordered set. A list of elements of type `aType` is specified as
  ```
  list of aType
  ```
  for example
  ```
  list of boolean
  ```

- *array* A fixed size list. An array of *n* elements of type `aType` is specified as
  ```
  array of n aType
  ```
  where n can be either an integer constant or a program parameter; for example
  ```
  array of 10 string
  array of ProgParm1 real
  ```

- *struct* A fixed-sized, ordered group of elements. Each element of the struct has a name and a specific type A struct is specified by `struct of` followed by a listing of the name and type of each of its components, with elements separated by a comma (,), as in:
  ```
  struct of type1 name1, type2 name2, ..., typeN nameN
  ```
  for example
  ```
  struct of integer red, integer green, integer blue
  ```

Figure 3: Aggregate Data Types

As with constants, data types can be arbitrarily complex. For example, the constant introduced in Section 2.4 could be described by the following datatype:

```
struct of
   array of 3 set of integer    intSetArray,
   set of set of string         stringSetSet,
   set of list of set of float  floatSetListSet,
   integer                      intComponent,
   string                       stringComponent;
```

Rather than replicating a lengthy type definition, one can give it a name in the types section, and use that name instead. This operation is called "renaming" of the defined type. Thus, if one defines

```
array3SetInt ≡ array of 3 set of integer;
```

`array3SetInt` is a renaming of `array of 3 set of integer`, and the datatype describing the constant from Section 2.4, can be described as:

```
struct of
     array3SetInt intSetArray,
     (rest of the definition unchanged);
```

4.3 Compatible Types

4.3.1 Assignment Compatibility

Two expressions are assignment compatible if one of the following is true:

1) they have the same type,

2) the type of either or both is a remaning of a common type,

3) one expression is of type float (or a renaming of it) and the other is of type integer (or a renaming of it).

Additionally, aggregate constant expressions are considered assignment compatible with other expressions if the elements of the constant expression are assignment compatible with the corresponding elements of the other expression. For example, given the following type definition:

```
struct of
     array of 3 integera,
     integer          b;
```

the following constant expressions are assignment compatible with expressions of this data type:

```
[ [ 0, 0, 0], 6 ]

[ [ 0, 0, 0.0], 6 ]

[ [ 0, 0, 0], 6.0 ]
```

while the following expressions are not:

```
[ 0, 0, 0, 6 ]

[ { 0, 0, 0 }, 6 ]

[ [ 0, 0 ], 6 ]
```

4.3.2 Ordering of Data Types.

In order for two expressions to be compered, it is necessary that they be of assignment compatible data types. Comparison of expressions which are sets, lists, or arrays is done element by element. Structures are compared field by field. Strings are compared character by character with the ordering based on the ASCII character set. For booleans, `true` is considered greater than `false`.

4.4 Formal Definition of the Types Section

The formal definition of a Data Types section is given in Figure 4, "BNF definition of the Types Section," on page 8. The token `<userDefinedTypeName>` refers to some previously defined type (`<Identifier>` token in the rule for *ATypeDef*).

5 Functions

To allow programs written in Swarm to interface with the outside world, and to simplify some operations which would be awkward in Swarm, support is provided for declaring and calling external functions from a Swarm program. These functions are declared in the functions section, and are invoked by placing a call to the function within an expression (see section 7 for additional details on calling functions). External functions must return a value (void functions are not permitted) which can be of any allowable data type. Parameters to the functions can also be of any

TypeSection ::= `types` *TypeDefList*

TypeDefList ::= *ATypeDef* | *TypeDefList ATypeDef*

ATypeDef ::= `<Identifier>` ≡ *AType* ;

AType ::= `boolean|integer|real|float|double|<userDefinedTypeName>`
| `list of` *AType*
| `set of` *AType*
| `struct of` *TypeList*
| `array of <Number>` *AType*
| `array of <Identifier>` *AType*

TypeList::= *AType* <Identifier>
| *TypeList* , *AType* `<Identifier>`

Figure 4: BNF definition of the Types Section

legal Swarm data type, and functions which accept no parameters are permitted. For example, a function *sum*, which takes a list of float values and returns a float, is declared as:

```
float sum(list of float v);
```

A function *CurrSet*, which takes no parameters and returns a set of integers, would be declared as

```
set of integer CurrSet();
```

These external functions are coded using the C programming language and combined with the compiled Swarm program at link time (see Appendix D for more details on producing an executable Swarm program). Parameters having simple types (integer, float, and boolean) are passed by value, parameters of complex types are passed by reference, with a `VAR_PTR` to the actual variable being used in the invokation. String parameters are passed as `char *`. The first example above has the following prototype when invoked:

```
double sum(VAR_PTR v)
```

When a complex value is returned from an external function, the function is actually called with an additional parameter which is a `VAR_PTR` to receive the result, and no value is returned by the function. Thus, the second example has the following prototype when invoked:

```
void CurrSet(VAR_PTR result)
```

The formal definition of the Function Declaration section is given in Figure 5, "BNF definition of the Function Declaration Section," on page 8. The rule for AType is defined in Section 4.

FunctionDeclaration ::= functions *FunctionDefList*

FunctionDefList ::= *AFunctionDef* | *FunctionDefList AFunctionDef*

AFunctionDef ::= *AType* `<Identifier>` (*TypedFormalList*) ;
| *AType* `<Identifier>` ( ) ;

TypedFormalList ::= *ATypedFormal* | *TypedFormalList* , *ATypedFormal*

ATypedFormal ::= *AType* `<Identifier>`

Figure 5: BNF definition of the Function Declaration Section

6 Definitions

The Definition Section provides a means for naming expressions and using those names in place of the corresponding expressions in the remainder of the program for clarity purpose. A macro can include a set of parameters that will be bound to appropriate values upon invoking the macro. A macro is specified as follows:

<macroName>(<param$_1$>, <param$_2$>, ..., <param$_N$>) ≡ <expression>;

where <expression> is any legal expression (See "Expressions" on page 9) containing only any of the formal parameters of the macro (<param$_i$>), constants, and any previously defined items (program argument, function, macro, tuples, ...). If the macro does not have any parameters, the parenthesis are omitted. The scope of a macro extends from the definition of the macro until the end of the program.

A macro is invoked by typing its name followed by a list of values which bind the formal parameters of the macro. The macro is then "replaced" by its corresponding expression where each instance of a formal parameter has been replaced by its value from the invokation.

The formal description of the Definition section is shown inFigure 6, "BNF definition of the Definitions Section," on page 9.

DefinitionSection ::= `definitions` *DefinitionList*

DefinitionList ::= *DefinitionList ADefinition* | *ADefinition*

ADefinition ::= <Identifier> *DefinitionFormals* ≡ *OrExpression* ;

DefinitionFormals ::= (*DefinitionFormalList*) | λ

DefinitionFormalList ::= *DefinitionFormalList* , <Identifier> | <Identifier>

Figure 6: BNF definition of the Definitions Section

7 Expressions

The Swarm language supports a rich variety of expression forms. This section provides first an overview of the various classes of expressions, followed by a more detailed description of the syntax and semantics of each class.

Expressions fall into three broad classes: arithmetic expressions, dataspace expressions, and constructors (or generators). The class of arithmetic expressions includes the common mathematical expressions, boolean expressions (arithmetic comparison, logical *and* and *or*), and a variety of expressions for manipulating the aggrate data types. Dataspace expressions include the class of expressions which query the dataspace and bind variables. Constructors are used to perform a single operation over an arbitrary number of operands using the Swarm three part notation. The complete grammar for expressions is given in Figure 7, "BNF definition of a Swarm expression," on page 10 and Figure 8, "BNF definition of a Swarm expression (Continued)," on page 11. The remainder of this section gives a detailed discussion of the various expressions.

7.1 Arithmetic Expressions

7.1.1 Numeric Expressions

The Swarm language supports all basic integer and floating point operators, namely + (binary), - (unary and binary),* , /, and `mod`. Figure 9, "Value of a Numeric Expression," on page 12, describes the effect of each operator. If any operand of a binary operator is a floating point number, both operands are promoted to floating point and the resulting value is a floating point.

Query ::= *OrExpression* | *Query* , *OrExpression*

OrExpression ::= *AndExpression* | *OrExpression* ∨ *AndExpression*

AndExpression ::= *TildeExpression* | *AndExpression* ∧ *TildeExpression*

TildeExpression ::= *RelationalExpression*
| *TildeExpression* ~ *RelationalExpression*
| *TildeExpression* ≈ *RelationalExpression*

RelationalExpression ::= *UnionExpression*
| *RelationalExpression RelOp UnionExpression*
| *RelationalExpression* ∈ *UnionExpression*
| *RelationalExpression* ∉ *UnionExpression*
| *RelationalExpression* ⊆ *UnionExpression*
| *RelationalExpression* ⊂ *UnionExpression*
| *RelationalExpression* ⊇ *UnionExpression*
| *RelationalExpression* ⊃ *UnionExpression*
| *RelationalExpression* ⊄ *UnionExpression*

RelOp ::= = | ≠ | < | ≤ | ≥ | >

UnionExpression ::= *AdditiveExpression*
| *UnionExpression* ∪ *AdditiveExpression*
| *UnionExpression* ∩ *AdditiveExpression*

AdditiveExpression ::=*MultiplicativeExpression*
| *AdditiveExpression* + *MultiplicativeExpression*
| *AdditiveExpression* - *MultiplicativeExpression*

MultiplicativeExpression ::= *UnaryExpression*
| *MultiplicativeExpression* * *UnaryExpression*
| *MultiplicativeExpression* / *UnaryExpression*
| *MultiplicativeExpression* `mod` *UnaryExpression*

UnaryExpression ::= *PostfixExpression*
| ¬ *UnaryExpression*
| - *UnaryExpression*
| `head` (*UnaryExpression*) | `tail` (*UnaryExpression*)
| `old` . *UnaryExpression*
| | *UnaryExpression* |
| `min` (*OrExpression* , *OrExpression*) | `max` (*OrExpression* , *OrExpression*)

Figure 7: BNF definition of a Swarm expression

Swarm also has the binary operators `min` and `max` which operands can be of any type as long as they are comparable (see Section 4.3). The cardinality operator | | can be applied to numeric expressions to compute the absolute value.

7.1.2 Set Expressions

The Swarm language supports the standard set operators, namely membership (∈), non-membership (∉), subset (⊆), proper subset (⊂), superset (⊇), proper superset (⊃) and not subset (⊄). Each of these operators returns a boolean value. Figure 10, "Truth Value of Set Expressions," on page 12, describes the types of the operands and the condition upon which a set expression returns true. Swarm supports also a cardinality operator (| |) which returns the

PostfixExpression ::= *Factor*
| *PostfixExpression* [*ExpressionList*]
| *PostfixExpression* [*ArrayReplacementList*]
| *PostfixExpression* (*OptionalExpressionList*)
| *PostfixExpression* . <Identifier>
| *PostfixExpression* †

Factor ::= <Identifier> | <Number> | <Boolean> | <String>
| ∞ | _ | _T_ | ∅
| [*OptionalExpressionList*] | { *OptionalExpressionList* }
| ⟨ *Constructor* ⟩
| (*OrExpression*)

ArrayReplacementList ::= *ArrayReplacement* | *ArrayReplacementList* ; *ArrayReplacement*

ArrayReplacement ::= *ExpressionList* \ *OrExpression*

Constructor ::= Σ *IdentifierList* : *Query* : : *OrExpression*
| ∏ *IdentifierList* : *Query* : : *OrExpression*
| **min** *IdentifierList* : *Query* : : *OrExpression*
| **max** *IdentifierList* : *Query* : : *OrExpression*
| # *IdentifierList* : : *Query*
| **SET** *IdentifierList* : *Query* : : *OrExpression*
| **ARRAY** *IdentifierList* : Query : : *OrExpression*
| ∀ *IdentifierList* : *ExpressionList* : : *Query*
| ∃ *IdentifierList* : *ExpressionList* : : *Query*

OptionalExpressionList ::= *ExpressionList* | λ

ExpressionList ::= *OrExpression* | *ExpressionList* , *OrExpression*

ActualParameters ::= *OrExpression* | *ActualParameters* , *OrExpression*

IdentifierList ::= *IdentifierDeclaration* | *IdentifierList* ; *IdentifierDeclaration*

IdentifierDeclaration ::= *AType IDList*

IDList ::= <Identifier> | *IDList* , <Identifier>

Figure 8: BNF definition of a Swarm expression (Continued)

number of elements in the set it is applied to.

7.1.3 List Expressions

The list operators in Swarm allows one to build a list or to access the content of a list. Figure 11, "Value of List Expressions," on page 13, describes the different operators.

7.1.4 Array Expressions

Swarm provides operators for examining and modifying the contents of array data. The index operator ([ ]) is used to retrieve the value stored at a position within an array, and the replacement operator ([\ ]) allows for items within an array to be modified. Note that the = operator is not used to change the value stored at a position within an array. This happens because arrays are always either completely bound or completely unbound; thus an expression of the form **a** = *OrExpression*, where **a** is an array will either bind **a** to the result of evaluating *OrExpression* (if

Expression	Value
`-` *op*	the negative of *op*
op_1 `+` op_2	the sum of op_1 and op_2
op_1 `-` op_2	the subtraction of op_2 from op_1
op_1 `*` op_2	the product of op_1 and op_2
op_1 `/` op_2	the division of op_1 by op_2. op_2 must not be equal to 0. Integer division results in the integer quotient whose magnitude is less than or equal to that of the true quotient, and with the same sign. Thus 9 / 4 = 2 while 9.0 / 4 = 9 / 4.0 = 9.0 / 4.0 = 2.25.
op_1 `mod` op_2	the remainder of the division of op_1 by op_2. op_2 must not be equal to 0. The remainder has the same sign as the *op*1, so that the expression $(op_1 / op_2) * op_2 + op_1 \bmod op_2$ corresponds to op_1.
`min(`op_1 `,` op_2`)`	the minimum of op_1 and op_2
`max(`op_1 `,` op_2`)`	the maximum of op_1 and op_2
`|` *op* `|`	the absolute value of *op*

Figure 9: Value of a Numeric Expression

Expression	op_1	op_2	Returns True when
$op_1 \in op_2$	<aType>	set of <aType>	op_1 is an element of op_2
$op_1 \notin op_2$	<aType>	set of <aType>	op_1 is not an element of op_2
$op_1 \subseteq op_2$	set of <aType>	set of <aType>	every element of op_1 belongs to op_2, and op_1 can be equivalent to op_2
$op_1 \subset op_2$	set of <aType>	set of <aType>	every element of op_1 belongs to op_2, but at least one element of op_2 does not belong to op_1
$op_1 \supseteq op_2$	set of <aType>	set of <aType>	every element of op_2 belongs to op_1, and op_2 can be equivalent to op_1
$op_1 \supset op_2$	set of <aType>	set of <aType>	every element of op_2 belongs to op_1, but at least one element of op_1 does not belong to op_2
$op_1 \not\subset op_2$	set of <aType>	set of <aType>	at least one element of op_1 does not belong to op_2

Figure 10: Truth Value of Set Expressions

`a` is unbound) or compare `a` to the result of evaluating *OrExpression* (if `a` is already bound).

Indexing into multidimensional can be done either in the style used in C, or by providing a list of expressions for the index value. For example, if `a` has the type `array of 10 array of 10 integer`, then the expressions `a[3,6]` and `a[3][6]` both extract the 7th element of the 4th array (as this example shows, arrays are based at 0). No checking of array bounds is done at compile time, all such checking is done at run-time. The compiler simply

Expression	op_1	op_2	Returns
List creation:			
$op_1 + op_2$	<aType>	list of <aType>	list op_2 prepended with element op_1
$op_1 + op_2$	list of <aType>	<aType>	list op_2 appended with element op_1
$op_1 + op_2$	list of <aType>	list of <aType>	list op_1 concatenated with list op_2
List access:			
op_1 [op_2]	list of <aType>	number	element op_2 of listop_1
head(op_1)	list of <aType>		first element of list op_1
tail(op_1)	list of <aType>		list op_1, without its first element
$\|op_1\|$	list of <aType>		number of elements in the list

Figure 11: Value of List Expressions

requires that index values be integral.

The replacement expression allows for (possibly multiple) value(s) to be inserted into an array which has already been bound by some other expression. Each replacement expression consists of 2 parts, an index (see the previous paragraph), and a new value. Multiple assignments can be done with a single replacement expression by listing multiple index/value pairs within the brackets, separated by semicolons. The replacement expression evalutates to an array with the new values installed.

Thus, if `A` is a variable having the type `array of 10 array of 15 set of integer`, then the following expression always evaluates to true, (and is occasionally substituted for true by the compiler).

```
7 ∈ A[4,8 \ { 7 }][4][8]
```

Figure 12, "Value of Array Expressions," on page 13, summarizes the array expressions.

Expression	op_1	op_2	op_3	Returns
Array access:				
op_1 [op_2]	list of <aType>	list of number		element op_2 of arrayop_1
Array replacement:				
op_1 [op_2 \ op_3]	list of <aType>	list of number	<aType>	array op_1 with value op_3 in element op_2

Figure 12: Value of Array Expressions

7.1.5 Structure Expressions

The only operation valid on structures is field extraction using the dot (.) operator. The right-hand side of the expression must be the name of some field within the structure type, and the result is an expression containing that value.

7.1.6 Relational Expressions

Swarm provides the standard relational operators, i.e., equality (=), inequality (≠), less than or equal (≤), strictly less than (<), greater than or equal (≥), and strictly greater than (>). Only elements of comparable types can be used as operands for these operators (See "Compatible Types" on page 7). The result of all these operators is a boolean indicating whether the relational expression holds or not.

The = operator has a more complex behavior than the other relational operators. If both operands are bound, the equality test checks that the values of the operand are equivaluent. If one of the two operands is not bound when the equality test is performed, the test always succeeds and, as a side effect of the test, the value of the unbound operand is set to the value of the bound operand. Both operands cannot be unbound.

7.1.7 Logical Expressions

Swarm provides 3 logical operators: unary negation (¬), binary and (∧), and binary or (∨). The operands of these operators must be of type Boolean (or a renaming of it). The result is of type Boolean. Figure 13, "Truth Value of Logical Expressions," on page 14, describes the truth value of a logical expression.

Expression	Returns true when
$\neg\, op$	op = false
$op_1 \wedge op_2$	op_1 = true and op_2 = true
$op_1 \vee op_2$	op_1 = true or op_2 = true

Figure 13: Truth Value of Logical Expressions

Currently, the compiler does not correctly handle logical expressions which consist of a disjunct of conjuncts, i.e. `a ∨ (b ∧ c ∧ d) ∨ e`. However, such an expression can be re-written by distributing the operators (which is most likely the way the compiler would ultimately do it anyways), so this is not necessarily a big problem.

7.2 Dataspace Expressions

The class of dataspace expressions includes queries which check for the presence of items within the dataspace (both tuples and synchrony elements), and an operator which allows deletions to be specified in the query, as allowed in the original specification of the Swarm notation.

7.2.1 Tuple Queries

A tuple query expression is a boolean expression which has the value `true` if there is a binding of the actual parameters to the tuple that can be satisfied by matching some item in the dataspace. The general form of a tuple query is `tuple_name(actual_parameters)`.The`tuple_name`mustmatchsometupleortransactiontypename, and each actual parameter can be any expression that is assignment compatible with the data types of the corresponding formal parameter. If any of the actual parameters are unbound variables, these variables will be bound if the query succeeds.

7.2.2 Synchrony Queries

Synchrony queries are the mechanism whereby the contents of the synchrony relation are queried. As with tuple queries, the result is a boolean expression indicating whether a synchrony entry satisfying the query was present. Synchrony queries can be used to bind variables.

7.2.3 Tuple Deletion

As a shortcut for specifying the deletion of a dataspace element found during a query, a dagger can be placed after any dataspace expression. The dagger does not affect the truth value of the query, and the deletion is only performed if the entire query succeeds. Daggers are not currently permitted in the query portions of constructors. That is, to find and delete all tuples of type `T(integer i)`, the following idiom should be used:

```
⟨ integer i : T(i) :: T(i)† ⟩
```

7.3 Constructors

A constructor expression uses the three-part notation to perform a single operation over a collection of objects that satisfies some predicate. The three-part notation ⟨ *op quantified_variables* : *range* :: *OrExpression* ⟩ used throughout the text is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. Each such instantiation of the variables is substituted in *OrExpression* producing a multiset of values to which *op* is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for *op*, e.g., `true` when *op* is ∀.

7.3.1 Arithmetic Constructors

Swarm provides four arithmetic constructor operators, sum (Σ), product (Π), minimum (`min`), and maximum (`max`). The *OrExpression* part of the constructor must have a numeric data type. Figure 14, "Zero Values of Arithmetic Constructor Operators," on page 15, lists the zero value of each of the operators.

Operator	Zero Value
Σ	0
Π	1
`min`	+∞
`max`	−∞

Figure 14: Zero Values of Arithmetic Constructor Operators

7.3.2 Count Constructor

The count constructor (#) is a bizarre creature. It's a three part notation with only two parts. The *OrExpression* which normally accompanies a constructor is implicitely 1 and therefore not needed. But because a colon looks stupid, a double colon is used to separate the variables from the query. The zero value of the count operator is 0. While the query of a count operator typically consists of a single dataspace expression, this is not required. However, if several expressions occur in the query, the count constructor evaluates to the number of ways the entire query can be bound. For example,

```
⟨ # integer i :: T(i), X(i) ⟩
```

does not evaluate to the sum of the number of `T` and `X` tuples in the dataspace, but rather to the number of elements with a common parameter. So, with a dataspace containing

```
T(1), T(2), T(3), T(4), X(2), X(4), X(6), X(8)
```

the value of the expression is 2, not 8.

7.3.3 Set Constructors

The set constructor (`SET`) is used to build a set based on the content of the dataspace. The zero value of this operator is the empty set. Using the dataspace from Section 7.3.2., the expression

```
⟨ SET integer i : T(i), X(i):: i ⟩
```

evaluates to

```
{ 2, 4 }
```

7.3.4 Array Constructors

The array constructor (`ARRAY`) is used to specify the content of an array at run-time. Unlike the other constructors, only range expressions are allowed in the query; and the range must evaluate to the indexes of the array. Only a single range can be specified in an array constructor. Thus, to fill a multi-dimensional array, you must use nested array constructor. For example, given the following declaration for tuple type `T`

```
T(array of 10 array of 15 integer a)
```

the following constructor can be used to create a tuple T with an array filled with integers from 1 to 150

```
T( ⟨ ARRAY integer i : 1 ≤ i ≤ 10::
   ⟨ ARRAY integer j : 1 ≤ j ≤ 15:: (i-1)*15+j ⟩ ⟩
   )
```

7.3.5 Logical Constructors

The logical constructor operators (∀ and ∃) are actually boolean operators in disguise. Thus, `true` is the zero element for ∀, and `false` for ∃. Referring again to the dataspace from Section 7.3.2., the following constructors evaluate to true and false, respectively

```
⟨ ∀ integer i,j : T(i), X(j) :: j = 2 * i ⟩
   ⟨ ∃ integer i,j : T(i), X(j) :: j ≠ 2 * i ⟩
```

8 Tuples and Transactions Declarations

Tuples and transactions declarations to specify the types of tuples and transactions that can be inserted in the dataspace. This is performed by specifying the range of possible values for each of the formal parameters.

The formal definition of a *Tuples* or *Transactions Declarations Section* is given in Figure 15, "BNF definition of a Tuples or Transactions Declarations Sections," on page 17. The rule for *TypedFormalList* is defined in Section 5, and for *ExpressionList* in Section 7.

9 Transaction Definitions

For each transaction declared in a Transaction Declaration Section, one needs to provide the corresponding computation in a Transaction Definitions Section. Each definition starts with the transaction name and its formal parameters, as listed in a Transaction Declaration Section, followed by a sequence of subtransactions. A subtransaction is composed of a query and an action. For more details on the Swarm model and the semantic associated with transactions, see [3] . Although the model places no requirement on the ordering of conjuncts within a query, the programming language requires that variables be bound (either by a dataspace query or by the = operator) prior to their use in any arithmetic expression.

The formal description of a Transaction Definitions Section is given in Figure 16, "BNF definition of the Transac-

SwarmTuples ::= **tuples types** *TupleDeclarations*

SwarmTransactions ::= **transactions types** *TupleDeclaration*

TupleDeclarations ::= *ATupleDeclaration* | *TupleDeclarations ATupleDeclaration*

ATupleDeclaration ::= **<Identifier>** ;
| **<Identifier>** (TypedFormalList) ;
| ⟨ *IdentifierList* : *ExpressionList* : : *TupleList* ⟩ ;
| ⟨ *IdentifierList* : : *TupleList* ⟩ ;

TupleList ::= *ATuple* | *TupleList* , *ATuple*

ATuple ::= **<Identifier>** | **<Identifier>** (*FormalParameters*)

FormalParameters ::= **<Identifier>** | *FormalParameters* , **<Identifier>**

Figure 15: BNF definition of a Tuples or Transactions Declarations Sections

tion Definitions section," on page 18. The rule for *ActualParameter* is defined in Section 7.

10 Swarm Initialization

The Swarm Initialization section lists the tuples, transactions, and synchrony relation entries present in the dataspace at the start of the execution of the program.

The formal definition of a Swarm Initialization Section is shown in Figure 17, "BNF definition of the Swarm Initialization section," on page 17. The rule for *ActualParameters* is given in Section 7.

InitializationSection ::= **initialization** *InitList*

InitList ::= *AnInit* | *InitList AnInit*

AnInit ::= *InitItem* ; | ⟨ *IdentifierList* : *ExpressionList* : : *InitItemList* ⟩ ;

InitItemList ::= *InitItem* | *InitItemList* , *InitItem*

InitItem ::= *InitActionItem* | *InitActionItem* ~ *InitActionItem*

InitActionItem::= **<Identifier>** | **<Identifier>** (*ActualParameters*)

Figure 17: BNF definition of the Swarm Initialization section

11 Visualization Spaces

For all the spaces specified in the header part of a visualization (see Section 3) except the animation space, one must define the tuple types that make each space. In the case of the input space, this corresponds to the tuple and transaction types defined in the Swarm program being visualized.

As shown in Figure 18, "BNF definition of the Space section," on page 18, a space section is composed of a space identifier followed by a set of tuple declarations, as described in Section 8. Multiple space definitions for the same space identifier are combined.

TransactionSection ::= `transactions definitions` *TransactionList*

TransactionList ::= *ATransaction* | *TransactionList ATransaction*

ATransaction ::= `<Identifier>` (*TransactionFormals*) ≡ *SubtransList* ;
| `<Identifier>` ≡ *SubtransList* ;

TransactionFormals::= `<Identifier>` | *TransactionFormals* , `<Identifier>`

SubtransList ::= *ASubTransaction* | *ASubTransaction MoreSubtransactions*

ASubTransactions ::= *IdentifierList* : *Query* → *TransactionActions*
| *Query* Æ *TransactionActions*
| *IdentifierList* : *SpecialPredicate* , *Query* → *TransactionActions*
| *IdentifierList* : *SpecialPredicate* → *TransactionActions*
| *SpecialPredicate* , *Query* → *TransactionActions*
| *SpecialPredicate* → *TransactionActions*
| *SubtransGenerator*

MoreSubtransactions ::= | | *SubtranList*
| *SubtransGenerator*
| *SubtransGenerator MoreSubtransactions*

SubtransGenerator ::= ⟨ | | *IdentifierList* : *ExpressionList* : : *SubtransList* ⟩

TransactionActions ::= `skip` | *TransActionList*

TransActionList ::= *ATransAction* | *TransActionList* , *ATransAction*
| *TransActionGenerator* | *TransActionList* , *TransActionGenerator*

ATransAction ::= *TransActionItem*
| *ATransAction* †
| *TransActionItem* ~ *TransActionItem*
| (*ATransAction*)

TransActionGenerator ::= ⟨ *IdentifierList* : *ExpressionList* : : *TransActionList* ⟩

TransActionItem ::= `<Identifier>` | `<Identifier>` (*ActualParameters*)

SpecialPredicate::= `AND` | `OR` | `NAND` | `NOR` | `TRUE`

Figure 16: BNF definition of the Transaction Definitions section

ASpaceDeclaration ::= `<SpaceName> space` *TupleDeclarations*

Figure 18: BNF definition of the Space section

12 Windows

The rendering of the animation space can take place in several independent windows. Each window has its own referential, coloring, viewing controls, etc... (see [2]). The Windows section allows the user to specify a set of rendering windows. The formal definition of the Window section is shown in Figure 19, "BNF definition of the Windows section," on page 19. The rule for *OrExpression* is defined in Section 7. `<WindowAttr>` is one of the window attributes listed in Appendix C: "Window Attributes".

WindowSection ::= **windows** *WindowDeclarations*

WindowDeclarations ::= *AWindowDeclaration* | *WindowDeclaration* ; *AWindowDeclaration*

AWindowDeclaration ::= **<Identifier>** (*WindowAttributeList*) | λ

WindowAttributeList ::= *AWindowAttribute* | *WindowAttributeList* , *AWindowAttribute*

AWindowAttribute ::= **<WindowAttr>** := *OrExpression*

Figure 19: BNF definition of the Windows section

13 Visualization Rules

A visualization rule specifies a logical relationship between two collections of tuples called the input and output spaces for the rule. The rule consists of a query and an action. The query is an arbitrary predicate which can include tests for the presence or absence of tuples in the rule's input space(s) and in the previous instances of its input and output spaces. The action consists of a list of tuples in the output space. The semantics of such a rule, given current and previous input spaces and previous output space are as follows: For every instantiation of variables such that the query is true, the corresponding tuples must be in the current output space. A mapping from an input space to an output space consists of one or more rules which map from the input to the output space. The output space for a mapping is defined as the union of the output spaces produced by each of the rules in the mapping.

The original model specifies a pipeline structure for the mappings. This has the effect of restricting rules to only a single input space, which must be the space immediately preceding the output space in the pipeline. By default, the compiler enforces this requirement. More relaxed semantics can be invoked by using the n option when running the compiler. Under this option, a rule can query any space before its output space in the pipeline. Use of this option can eliminate the need for copying tuples unchanged from space to space, although it reduces the compiler's ability to detect mistakes in the visualization program.

The formal definition of the Visualization Rules section is given in Figure 20, "BNF definition of the Visualization Rules section," on page 19. The rule for *ActualParameters* is defined in Section 7 and for *AnimationProducts* in Section 14.

RuleDeclaration ::= **rules** *Rules*

Rules ::= *ARule* | *Rules ARule*

ARule ::= **<Identifier>** ≡ *IdentifierList* : *Query* ⇒ *Actions* ;
| **<Identifier>** ≡ *Query* ⇒ *Actions* ;

Actions ::= *VisActions* | *AnimationProducts*

VisActions ::= *AVisAction* | *VisAction* , *AVisAction*

AVisAction ::= **<Identifier>** | **<Identifier>** (*ActualParameters*)

Figure 20: BNF definition of the Visualization Rules section

14 Graphical objects and functions for the animation space

The final visualization space is the animation space which contains a set of graphical objects to render in the windows. Each object is defined in terms of a collection of attributes such as color or position. An animation object is created as the result of some visualization rule but, as opposed to other spaces' objects, an animation object's attrrib-

utes have to be named, can be omitted (each has a default value), and can be specified in any order. The list of animation objects with their respective attributes is shown in Appendix A: "Graphical objects and their attributes".

In addition, as opposed to other spaces where objects have a binary existence, animation objects have a "lifetime" along which any of their attributes can be modified according to a user-specified transformation. Appendix B: "Time functions for graphical objects" describes the different functions that can be applied.

The formal definition of the Animation Elements is given in Figure 21, "BNF definition of the Animation Elements," on page 20. `<Window>` is a window identifer as described in Section 12, `<AnimationProduct>` is one of the graphical objects and `<AnimationAttribute>` one of the object's attributes as shown in Appendix A: "Graphical objects and their attributes".

AnimationProducts ::= *AnAnimationProduct* | *AnimationProduct* , *AnAnimationProduct*

AnAnimationProduct ::= `<AnimationProduct>`
| `<AnimationProduct>` (*AnimationAttributes*)

AnimationAttributes ::= *AnAnimationAttribute* | *AnimationAttribute* , *AnAnimationAttribute*

AnAnimationAttribute ::= `<AnimationAttribute>` := *OrExpression*
| `<AnimationAttribute>` := *AnimationFunction*
| `<Window>` := `<Identifier>`
| `<Window>` := `<Number>`

AnimationFunction ::= `_step` (*OrExpression* , *OrExpression* , *OrExpression*)
AnimationFunctionTail
| `_ramp` (*OrExpression* , *OrExpression* , *OrExpression* , *OrExpression*)
AnimationFunctionTail
| `_constant` (*OrExpression* , *OrExpression* , *OrExpression*)
AnimationFunctionTail
| `_square` (*OrExpression* , *OrExpression* , *OrExpression* ,
OrExpression , *OrExpression* , *OrExpression*)
AnimationFunctionTail

AnimationFunctionTail ::= `then` *AnimationFunction* | λ

Figure 21: BNF definition of the Animation Elements

15 Visualization Initialization

The Visualization Initialization section is almost identical to the Swarm Initialization section with the addition that animation objects can be included. Thus, in the formal definition given in Section 10, the rule for InitItem if replaced by the one shown in Figure 21, "BNF definition of the Animation Elements," on page 20. The rule for AnAnimationProduct is defined in Section 14.

InitItem ::= *InitActionItem*
| *InitActionItem* ~ *InitActionItem*
| *AnAnimationProduct*

Figure 22: BNF definition of the Visualization Initialization section

Appendix A: Graphical objects and their attributes

The following graphical objects are provided by the interpreter. All objects have a lifetime attribute, which is of type `list of 2 numbers`. The type `coord` is an alias for `array of 3 float` and specifies the X/Y/Z coordinates of the point. The type `color` is `array of 3 integer` and specifies the red/green/blue color values, each in the range 0 to 255. The default color is *white*, or [255, 255, 255].

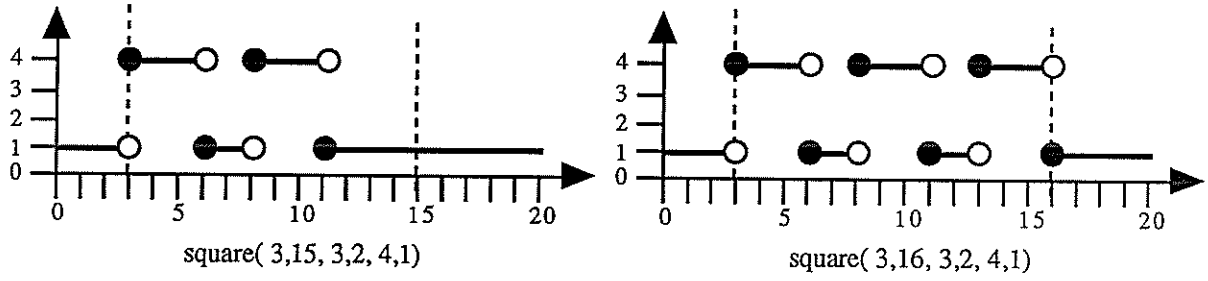
Object Type	Attribute	Type	Default
_box	_color	color	white
	_corner	coord	[0, 0, 0]
	_fill	boolean	false
	_fillcolor	color	_color
	_width	number	1
	_xrot	number	0
	_xsize	number	1
	_yrot	number	0
	_ysize	number	1
	_zrot	number	0
	_zsize	number	1
_circle	_center	coord	[0, 0, 0]
	_color	color	white
	_fill	boolean	false
	_fillcolor	color	_color
	_radius	number	1
	_width	number	1
	_xrot	number	0
	_yrot	number	0
	_zrot	number	0
_crect	_center	coord	[0, 0, 0]
	_color	color	white
	_fill	boolean	false
	_fillcolor	color	_color
	_width	number	1
	_xrad	number	1
	_xrot	number	0
	_yrad	number	1
	_yrot	number	0
	_zrot	number	0
_label	_color	color	white
	_position	coord	[0, 0, 0]
	_text	string	""
_line	_arrowfrom	boolean	false
	_arrowsize	number	1
	_arrowto	boolean	false
	_chop	number	0
	_color	color	white
	_from	coord	[0, 0, 0]
	_to	coord	[0, 0, 0]
	_width	number	1

Object Type	Attribute	Type	Default
_octahedron	_center	coord	[0, 0, 0]
	_color	color	white
	_radius	number	1
_point	_color	color	white
	_position	coord	[0, 0, 0]
_polygon	_color	color	white
	_fill	boolean	false
	_fillcolor	color	_color
	_vertices	list of coord	[[0, 0, 0]]
	_width	number	1
_rectangle	_color	color	white
	_corner	coord	[0, 0, 0]
	_fill	boolean	false
	_fillcolor	color	_color
	_width	number	1
	_xrot	number	0
	_xsize	number	1
	_yrot	number	0
	_ysize	number	1
	_zrot	number	0
_shapecylinder	_color	color	white
	_facefill	boolean	true
	_facefillcolor	color	_color
	_fill	boolean	true
	_fillcolor	color	_color
	_origin	coord	[0, 0, 0]
	_width	number	1
	_xrot	number	0
	_yrot	number	0
	_zrot	number	0
	_zsize	number	1
_sphere	_center	coord	[0, 0, 0]
	_color	color	white
	_radius	number	1
_vector	_color	color	white
	_direction	triple	[0, 0, 0]
	_length	number	0
	_origin	coord	[0, 0, 0]
	_width	number	1

Appendix B: Time functions for graphical objects

Function	Start Time	End Time	Value Before	Value During	Value After
step(t,v_0,v_1)	t	t	v_0	N/A	v_1
ramp(t_0,v_0,t_1,v_1)	t_0	t_1	v_0	linear interpolation from v_0 at t_0 to v_1 at t_1	v_1
constant(t_0,v,t_1)	t_0	t_1	v	v	v
square($t_0,t_1,p_{on},p_{off},v_{on},v_{off}$)	t_0	t_1	v_{off}	square wave: v_{on} for p_{on} ticks, v_{off} for p_{off} ticks	v_{off}

The square function takes value v_{on} at time t_0, then alternates between v_{on} and v_{off} for the rest of the during period. All v_{on} periods last a complete time p_{on}; if the interval remaining in the during period is insufficient for a complete v_{on}, the value will be held at v_{off} until the expiration of the during period. The diagram below gives some examples of this for clarification. Both graphs show a square wave with $p_{on} = 3$ and $p_{off} = 2$. In the left graph the last v_{on} period ends at tick 11; if another period were started, it would begin at time 13 and end at time 16, after the expiration of the during. In the right graph there is sufficient time for an addition v_{on} period to be included.



Appendix C: Window Attributes

Attribute Name	Type	Default	Meaning
Position of the window			
`_WinPositionX`	integer	0	Horiz coord of the upper left corner
`_WinPositionXMin`	integer	0	Minimum horizontal position
`_WinPositionXMax`	integer	1280	Maximum horizontal position
`_WinPositionY`	integer	0	Vertical coordinate of the upper left corner of the window
`_WinPositionYMin`	integer	0	Minimal vertical position
`_WinPositionYMax`	integer	1024	Maximum vertical position
`_WinPositionChange`	winEnum†	_WinAllChange	Can the window be moved?
Size of the window			
`_WinSizeX`	integer	640	Width of the window
`_WinSizeXMin`	integer	0	Minimum width
`_WinSizeXMax`	integer	1280	Maximum width
`_WinSizeY`	integer	480	Height of the window
`_WinSizeYMin`	integer	0	Minimum height
`_WinSizeYMax`	integer	1024	Maximum height
`_WinSizeChange`	winEnum†	_WinAllChange	How can the window be resized?
Style of the window			
`_WinDisplayControl`	boolean	true	Should the control window be shown?
`_WinDisplayIconic`	boolean	false	Should the display start in iconic form?
`_WinDisplayLighting`	boolean	true	Should lighting be used in rendering?
Background of the window			
`_WinColorRed`	0...255	0	Red component of background
`_WinColorGreen`	0...255	0	Green component of background
`_WinColorBlue`	0...255	0	Blue component of background
`_WinHSVColorModel`	boolean	false	Use Hue-Saturation-Value model instead of RGB. A color value becomes: [Hue, Saturation, Value]
`_WinColorChange`	winEnum†	_WinAllChange	Can background be changed?
Viewing position			
`_WinCenterX`	integer	0	X coordinate of the center of vision
`_WinCenterY`	integer	0	Y coordinate of the center of vision
`_WinCenterZ`	integer	0	Z coordinate of the center of vision
`_WinCenterExp`	integer	0	$\log_{10}$ of coordinate factor‡
`_WinCenterScrollExp`	integer	0	$\log_{10}$ of scrolling factor‡
`_WinCenterChange`	winEnum†	_WinAllChange	Can the center be moved?
Azimuth of viewer's eye			
`_WinAzimuth`	0...360	0	Azimuth value
`_WinAzimuthChange`	winEnum†	_WinAllChange	Can the azimuth be changed?
`_WinAzimuthSpin`	integer	0	Cyclic increment or decrement
`_WinAzimuthSpinChange`	winEnum†	_WinAllChange	Can the azimuth spin be changed?

Attribute Name	Type	Default	Meaning
Depth of vision			
`_WinDepth`	integer	20	Depth of rendered area
`_WinDepthMin`	integer	1	Minimun depth
`_WinDepthMax`	integer	100	Maximum depth
`_WinDepthExp`	integer	0	$\log_{10}$ of depth factor‡
`_WinDepthChange`	winEnum†	_WinAllChange	Can the depth be changed?
Distance of viewer's eye			
`_WinDistance`	integer	10	Distance of viewer's eye
`_WinDistanceMin`	integer	1	Minimum distance
`_WinDistanceMax`	integer	50	Maximum distance
`_WinDistanceExp`	integer	0	$\log_{10}$ of distance factor‡
`_WinDistanceChange`	winEnum†	_WinAllChange	Can the distance be changed?
Incidence of viewer's eye			
`_WinIncidence`	-90...270	0	Incidence value
`_WinIncidenceChange`	winEnum†	_WinAllChange	Can the incidence be changed?
`_WinIncidenceSpin`	integer	0	Cyclic increment or decrement
`_WinIncidenceSpinChange`	winEnum†	_WinAllChange	Can the incidence spin be changed?
Incidence of viewer's eye			
`_WinRoll`	0...360	0	Roll value
`_WinRollChange`	winEnum†	_WinAllChange	Can the roll be changed?
`_WinRollSpin`	integer	0	Cyclic increment or decrement
`_WinRollSpinChange`	winEnum†	_WinAllChange	Can the roll spin be changed?

† This data type contains the following values:

`_WinAllChange`	Any modification is allowed
`_WinNoChange`	No modification is allowed
`_WinXChange`	Only horizontal modifications are allowed (`_WinPositionChange` and `_WinSizeChange` only)
`_WinYChange`	Only vertical modifications are allowed (`_WinPositionChange` and `_WinSizeChange` only)
`_WinKeepAspect`	Maintain X/Y ratio of the window dimensions (`_WinSizeChange` only)

‡ When an exponent is used in conjunction with some base value, the actual value is $base*10^{exp}$

Appendix D: Swarm Character Set

#	Char	Keys	Glyph
32	space	space	
33	!	!	
34	"	"	
35	#	#	
36	$	$	
37	%	%	
38	&	&	
39	'	'	
40	(	(	
41	)	)	
42	*	*	
43	+	+	
44	,	,	
45	-	-	
46	.	.	
47	/	/	
48	0	0	
49	1	1	
50	2	2	
51	3	3	
52	4	4	
53	5	5	
54	6	6	
55	7	7	
56	8	8	
57	9	9	
58	:	:	
59	;	;	
60	<	<	
61	=	=	
62	>	>	
63	?	?	
64	@	@	
65	A	A	
66	B	B	
67	C	C	
68	D	D	

#	Char	Keys	Glyph
69	E	E	
70	F	F	
71	G	G	
72	H	H	
73	I	I	
74	J	J	
75	K	K	
76	L	L	
77	M	M	
78	N	N	
79	O	O	
80	P	P	
81	Q	Q	
82	R	R	
83	S	S	
84	T	T	
85	U	U	
86	V	V	
87	W	W	
88	X	X	
89	Y	Y	
90	Z	Z	
91	[	[	
92	\	\	
93	]	]	
94	^	^	
95	_	_	
96	`	`	
97	a	a	
98	b	b	
99	c	c	
100	d	d	
101	e	e	
102	f	f	
103	g	g	
104	h	h	
105	i	i	

#	Char	Keys	Glyph
106	j	j	
107	k	k	
108	l	l	
109	m	m	
110	n	n	
111	o	o	
112	p	p	
113	q	q	
114	r	r	
115	s	s	
116	t	t	
117	u	u	
118	v	v	
119	w	w	
120	x	x	
121	y	y	
122	z	z	
123	{	{	
124	|	|	
125	}	}	
126	~	~	
127			
128	℘	Op u/A	
129	A	Op A	
130	X	Op C	
131	∇	Op e/E	
132	⊑	Op n/N	
133	□	Op u/O	
134	⊏	Op u/U	
135	⌈	Op e/a	
136		Op `/a	
137	↓	Op i/a	
138	↑	Op u/a	
139		Op n/a	
140	α	Op a	
141	χ	Op c	
142	ε	Op e/e	

#	Char	Keys	Glyph
143		Op `/e	
144	'	Op i/e	
145		Op u/e	
146	⌊	Op e/i	
147		Op `/i	
148	∘	Op i/i	
149	⊥	Op u/i	
150	ν	Op n/n	
151	⌈	Op e/o	
152	ⓣ	Op `/o	head
153	•	Op i/o	
154	⊕	Op u/o	
155	⊖	Op n/o	
156	⌋	Op e/u	
157	↛	Op `/u	tail
158		Op i/u	
159	υ	Op u/u	
160	τ	Op t	
161	∏	Op *	*
162	∀	Op 4	\A
163	∃	Op 3	\E
164	∧	Op 6	/\
165	∞	Op 8	inf
166	∨	Op 7	\/
167	σ	Op s	
168	ρ	Op r	
169	γ	Op g	
170	‖	Op 2	||
171	E	Op E	
172	Y	Op U	
173	≠	Op =	<>
174	»	Op "	
175	O	Op O	
176	×	Op 5	
177	∑	Op +	\+
178	≤	Op ,	<=
179	≥	Op .	>=
180	ψ	Op y	

#	Char	Keys	Glyph
181	μ	Op m	
182	δ	Op d	
183	ω	Op w	
184	Π	Op P	
185	π	Op p	
186	β	Op b	
187	⊂	Op 9	\<
188	⊃	Op 0	\>
189	ζ	Op z	
190	†	Op '	@
191	o	Op o	
192	∉	Op ?	
193	¬	Op 1	!
194	λ	Op l	
195	≺	Op v	
196	ϕ	Op f	
197	ξ	Op x	
198	ι	Op j	
199	∩	Op \	
200	∪	Op |	
201	...	Op ;	
202			
203		Op `/A	
204		Op n/A	
205		Op n/O	
206	Θ	Op Q	
207	θ	Op q	
208	→	Op -	->
209	≡	Op _	==
210	“	Op [	
211	”	Op {	
212	–	Op]	
213	—	Op }	
214	∈	Op /	.?
215	≮	Op V	
216	∴	Op u/y	
217	⇐	Op u/Y	
218	≈	Op !	~~

#	Char	Keys	Glyph
219	‖	Op @	
220	⇒	Op #	=>
221	↦	Op $	
222	ø	Op %	{}
223	⊄	Op ^	
224	⊄	Op &	
225	⊆	Op (	.<=
226	⊇	Op)	
227	Ω	Op W	
228	P	Op R	
229	M	Op M	
230		Op i/E	
231	Ψ	Op Y	
232		Op u/E	
233		Op `/E	
234	Σ	Op S	
235	Δ	Op D	
236	Φ	Op F	
237		Op `/I	
238	H	Op H	
239	I	Op J	
240	K	Op K	
241	Λ	Op L	
242	«	Op :	
243		Op i/U	
244		Op `/U	
245	B	Op B	
246	⇔	Op I	
247	N	Op N	
248	⟨	Op <	<<
249	⟩	Op >	>>
250	η	Op h	
251	κ	Op k	
252	Z	Op Z	
253	Γ	Op G	
254	Ξ	Op X	
255	T	Op T	

Appendix E: Sample Swarm program

The following is an implementation of the Floyd-Warshall all-pairs shortest path problem. Given an undirected graph with a weight *w(i,j)* associated with each edge, the algorithm produces the length of the shortest path between each pair of nodes. We assume that *w(i,i)* = 0 for all nodes *i*, and *w(i,j)* is infinity for any nodes *i* and *j* that are not connected by an edge. The graph instance (including the weight function) is supplied through external functions which use the program's argument (*GraphId*) to select among various instances.

The algorithm operates on a two-dimensional array *dist*, here represented by a Swarm tuple of type *dist* having three components (the two array indices and the value). The core of the algorithm is the "scanning" of a node *k* which results in the updating of all *dist* tuples in parallel. This operation is performed by the subtransaction generator in the *Scan(k)* transaction. This transaction scans each node *k* in numeric order, starting with node 0.

```
program AllPairs(integer GraphId);

types
   distance ≡ real;

functions
   integer nnodes(integer graph);
   distance w(integer graph, integer i, integer j);

definitions
   range(x) ≡ 1 ≤ x < nnodes(GraphId);

tuple types
   ⟨ integer i, j; distance v :
      range(i), range(j) ::
         dist(i,j,v) ⟩;

transaction types
   ⟨ integer k : range(k) :: Scan(k) ⟩;

transaction definitions
   Scan(k) ≡
         k < nnodes(GraphId)
      →
         Scan(k+1)
   ||
      ⟨ || integer i,j : range(i), range(j) ::
         distance dij, dik, dkj :
         dist(i,j,dij), dist(i,k,dik),
         dist(k,j,dkj), dik + dkj < dij
         →
         dist(i,j,dij)†, dist(i,j,dik+dkj)
      ⟩;

initialization
   ⟨ integer i, j: range(i), range(j) :: dist(i,j,w(i,j)) ⟩;
   Scan(0);

end
```

Appendix F: Sample visualization program

The following is a visualization of the program in Appendix B. This rather trivial visualization maps each non-infinite distance into a box whose X-Y position is proportional to the distance and whose color and Z-size are both functions of the distance. The color is produced by the external function *vtocolor*, but all the position and size transforms are produced using macro definitions. This visualization uses two mappings with an intermediate space, named the "rutabaga" space (for no particular reason other than to emphasize that these names are totally arbitrary).

```
visualization AllPairs(integer GraphId)
   state => rutabaga => animation;

types
   distance ≡ real;

functions
   array of 3 integer vtocolor(distance v);
   integer nnodes(integer graph);
   distance infinity();

definitions
   range(x) ≡ 1 ≤ x < nnodes(GraphId);
   XYSIZE ≡ 0.9;
   XPOS(I) ≡ ((I)-(N/2.0));
   YPOS(I) ≡ ((N/2.0)-(I));
   ZSIZE(I) ≡ (I);

windows
   defaultwindow(_WinIncidence := 90);

state space
   ⟨ integer i, j; distance v :
      range(i), range(j) ::
         dist(i,j,v) ⟩;

rutabaga space
   finitebox(float xp, float yp, float zs,
         array of 3 integer the_boxes_color);

rules

get_the_boxes ≡
   integer i,j; distance v :
      dist(i,j,v), v < infinity()
   ⇒
      finitebox(XPOS(i),YPOS(j),ZPOS(v),vtocolor(v));

draw_the_boxes ≡
   float x,y,z; array of 3 integer c:
      finitebox(x,y,z,c)
   ⇒
      _rectangle(_window := defaultwindow,
         _fill := true, _corner := [x,y,z],
         _xsize := SIZE, _ysize := SIZE, _color := c);
```

```
end
```

Appendix G: External functions for the sample programs

Our sample Swarm program and visualization make use of the following four external functions:

```
distance infinity();
integer nnodes(integer graph);
distance w(integer graph, integer i, integer j);
array of 3 integer vtocolor(double v);
```

These functions must be written by the user in C (or C++) and included with the Swarm or visual code when it is compiled (see Appendix E). The code would have the following general structure.

```
// SKernel.h contains the definitions needed for variables
// (objects of type VAR and VAR_PTR), which vtocolor uses
#include "SKernel.h"

// limits.h contains the hardware limits of the machine; we
// use it to get our "infinity".  Since the Swarm code may add
// two infinities, we divide the maximum possible double by 2.
#include <limits.h>
#define INFINITY (DBL_MAX/2.0)

// Our first function is declared as "real infinity()"
// ("distance" is a type-name for "real").  Since this function
// returns a Pavane simple type (i.e., integer, real, or boolean)
// its C declaration parallels its Pavane declaration, with the
// necessary type conversions.  Pavane's "real" is C's "double".

double infinity(void) { return INFINITY; }

// "integer nnodes(integer graph)"
// Again, a simple type, so the C declaration parallels the
// Pavane.  "integer" is a C "long".

long nnodes(long graph) {
   // Some code here which determines the number of
   // nodes in the graph and returns it.  For example,
   // the graphs might be stored in files indexed by the
   // graph id; in this case the appropriate file would
   // be opened and read.
}

/* "real w(integer graph, integer i, integer j)"
 * As in the previous functions, the C parallels the Pavane.
 */

double w(long graph, long i, long j) {
   // Once again we somehow access the appropriate graph and
   // find the edge weight from i to j.  If i == j, we return
   // 0.0; if there is no edge from i to j, we must return
   // INFINITY.
}
```

```
// "array of 3 integer vtocolor(real v)"
// Here we have a return value which is a complex type.  In these
// cases, a pointer to the variable (VAR_PTR) is passed as the
// first argument to the C function and we have to return the
// value through it.  In other words, if the return value is a
// complex type, the C function has type "void" and gets an
// extra "VAR_PTR" argument as its first argument.

void vtocolor(VAR_PTR out_array, double v) {
long red, green, blue;
VAR_PTR elements;
   // The code to calculate values for red, green, and blue
   // (each an integer between 0 and 255) is omitted.  Once
   // the values are calculated, we make out_array into a
   // three-element array whose elements are red, green, and
   // blue in that order.

   // When an array is created, we must first allocate the
   // storage for the elements.  ArrayAllocate does this:
   elements = ArrayAllocate(3);

   // Next we bind the three elements to the previously-
   // calculated red, green, and blue values.  VarBindLong
   // takes a variable (actually a pointer to the variable)
   // and makes it into a Pavane integer with value as
   // given by the second argument.
   VarBindLong(&elements[0], red);
   VarBindLong(&elements[1], green);
   VarBindLong(&elements[2], blue);

   // Finally, the VarBindArray call is used to put all the
   // pieces together.  The three arguments are the variable
   // that is to be bound to the array, the size of the array,
   // and the elements that we just allocated and bound.
   VarBindArray(out_array, 3L, elements);
}
```

Appendix H: Producing executable Swarm programs and visualizations

This process is performed in three steps:

1)The swarm program or the visualization are compiled, generating a C file

2)The C file is compiled, generating some object file

3)The object file is combined with other object files, defining the external routines, and the appropriate run-time libraries.

More details are available on the machines on which Pavane is installed.

Bibliography

[1] Cox, K. C., "SwarmView Animation Vocabulary and Interpretation," Washington University, Department of Computer Science, Technical Report 91-10, 1991.

[2] Plun, J. Y., "Pavane User's Manual," Washington University, Dept of Computer Science, Technical Memo WUCS-TM-92-01, 1992.

[3] Roman, G.-C., and Cunningham, H. C., "Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1361-1373, 1990