Report Number: WUCS-91-08

1990-12-01

SwarmExec: A Prolog-Based Execution Engine for a Shared-Database Language with Visualization Capabilities

Authors: Kenneth C. Cox, C. Donald Wilcox, and Jerome Y. Plum

We have implemented a Prolog execution engine for the shared-database language Swarm extended with visualization capabilities. We call this execution engine SwarmExec. SwarmExec runs on a Macintosh Iifx under Advanced A.I. Systems' Prolog (AAIS) and communicates over an Ethernet connection with a Silicon Graphics Personal Iris which serves as a graphical engine and renders the visualizations. This paper describes the major design elements of SwarmExec. A basic familiarity with Swarm and its visualization extensions is assumed; the interested reader is referred to the referenced papers.

SwarmExec:
A Prolog-Based Execution Engine for a Shared-Dataspace Language with Visualization Capabilites

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

WUCS-91-08

December 1990

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

1 Introduction

We have implemented a Prolog execution engine for the shared-dataspace language *Swarm* extended with visualization capabilities. We call this execution engine *SwarmExec*. SwarmExec runs on a Macintosh® IIfx under Advanced A. I. Systems' Prolog (AAIS) and communicates over an Ethernet connection with a Silicon Graphics Personal Iris® which serves as a graphical engine and renders the visualizations. This paper describes the major design elements of SwarmExec. A basic familiarity with Swarm and its visualization extensions is assumed; the interested reader is referred to the referenced papers.

Section 2 of this paper briefly discusses the translation of Swarm programs into the form used by SwarmExec, called *SwarmProlog*; this topic will be discussed more completely in a forthcoming paper by Plun and Wilcox on the implementation of a graphical editor and compiler for the language. Section 3 discusses the execution of SwarmExec and the manner in which the Swarm transactions are executed. Section 4 describes the interface between SwarmExec and the Macintosh user and the Ethernet interface with the Personal Iris graphical engine.

2 Translation of Swarm into SwarmProlog

All Swarm language components are translated into Prolog before execution by SwarmExec. We refer to the translated form of these components as SwarmProlog, since (although they are simply Prolog) the SwarmProlog elements require the framework of SwarmExec to be executed. Three main components of the Swarm language are represented in SwarmProlog: the Swarm dataspace and visualization spaces, the Swarm transaction definitions, and the visualization mappings.

2.1 Dataspaces

Swarm, as extended with visualization capabilities, uses several distinct dataspaces (collections of tuples) representing the computation state. These spaces include the *tuple* space, the *transaction* space and the *synchrony relation* of Swarm, and the *proof*, *previous proof*, *object*, *previous object*, and *animation* spaces of the visualization extension.

Each of these spaces is represented by a collection of facts in the Prolog database. Each fact has the form

```
space_name( tuple_type( tuple_parameters ) ).
```

where `space_name` is one of `tuple`, `proof`, etc., `tuple_type` is the Swarm tuple type, and `tuple_parameters` is the parameter list (if any) of the particular tuple. Appropriate syntactic adjustments must be made to accommodate Prolog syntax; for example, the `tuple_type` and all other atoms should begin with small letters, as capital letters are used as variables in Prolog.

A Prolog goal with the name `initialize_dataspace` must be provided as part of the completed SwarmProlog program. The function of this goal is to generate the initial contents of the dataspaces, using the Prolog `assert` goal.

2.2 Transaction Definitions

A Swarm transaction consists of a collection of parallel subtransactions, each consisting of a list of variables, a query part, and an action part:

transaction(parameters) ≡
    $variables_1$: $query_1$ → $action_1$
|| $variables_2$: $query_2$ → $action_2$
    ...
|| $variables_n$: $query_n$ → $action_n$

Such a transaction definition is translated into a Prolog goal of the form:

```
transaction( parameters ) :-
    translation of subtransaction 1,
    translation of subtransaction 2,
        ...
    translation of subtransaction n.
```

Each subtransaction translation has the form

```
setquerytype( type ),
    (   translation of query,
        !,
        translation of action,
        itsucceeded
    ;
        itfailed
    )
```

The `setquerytype` goal, defined by the SwarmExec code, selects the type of the subtransaction; the type for a simple or local subtransaction is `local`, the type for a **NOR** subtransaction is `nor`, and so on. The remainder of the translation consists of a Prolog alternative construct; either the translation of the query succeeds, in which case the translation of the action and the goal `itsucceeded` are executed, or the translation of the query fails, in which case the goal `itfailed` is executed. `itsucceeded` and `itfailed` are goals defined by the SwarmExec code.

The variables are represented by Prolog variables. Care must be taken with variable names, since the scope rules of Swarm and Prolog differ (in Swarm the variable scope is confined to its subtransaction; in Prolog, it applies to the entire goal).

The translation of the query is quite direct; for example, to determine if the tuple *t(1)* is in the dataspace, the Prolog goal `tuple(t(1))` would be used. Queries for the presence of transactions are translated similarly; the goal `sw_doubletilde(`t_1`,`t_2`)` is used to query the closure of the synchrony relation. The action is also directly translated; the utility goals `insert_tuple`, `insert_trans`, and `insert_sync` are provided for inserting dataspace elements, and the goals `delete_tuple`, `delete_trans`, and `delete_sync` are provided for deletion. Various other utility goals to perform such operations as generators, summations, products, universal and existential quantifiers, and so forth are also provided and can be used in either the query or action parts of the translated goal; the implementation of these goals is discussed in section 3.

2.3 Visualization Mappings

A visualization consists of three distinct mappings (collections of visualization rules), the *proof* mapping, the *object* mapping, and the *animation* mapping. Each mapping transforms one or more input spaces to an output space; for example, the proof mapping transforms the state space (the Swarm dataspace) into the object space. A visualization rule consists of a list of variables, a query over one or more spaces, and a list of tuples to be produced:

variables : query $\Rightarrow$ production

The interpretation of such a rule is as follows: For all instantiations of the variables such that the query is true, include the list of tuples in the production in the output space This is exactly the same definition as a Swarm generator, except that queries over the dataspace are not permitted in a Swarm generator; the same translation mechanism suffices. Specifically, a rule such as the above would be translated into three Prolog goals:

```
name :- sw_visualization_rule([variables],
variables^query_goal(variables), production_goal).
```

```
query_goal(variables) :- translation of query.

production_goal(variables) :- translation of production.
```

The name for the first goal is arbitrarily chosen; its purpose is to identify the goal for the mapping list, as described below. The `sw_visualization` goal, a part of SwarmExec, finds all instantiations of the variable list which satisfy the `query_goal`; for each such instantiation, the `production_goal` is executed. The additional syntactic elements (such as the carat between the `variables` and the `query_goal`) are required by the underlying Prolog mechanisms, as discussed in section 3.3.

In addition to the rule translations, the collection of rules which make up each mapping must be represented. This is achieved by providing lists of all the rule names; for example, the proof mapping is specified by a Prolog fact

```
vis_rules( proof( [name1, name2, ...] ) ).
```

where each of the $name_i$ is the name of one of the rules making up the mapping.

3 Internal Mechanics of SwarmExec

SwarmExec consists of a collection of Prolog goals which, when provided with appropriate SwarmProlog, "executes" the SwarmProlog program. To use SwarmExec, Prolog is started on the Macintosh; the file containing SwarmExec is then consulted, after which the file(s) containing the SwarmProlog are consulted. (A more convenient interface for starting SwarmExec and consulting user programs is contemplated.)

3.1 Primary Control

The `swarm_run` goal is executed following loading of the files containing the SwarmProlog goals. All remaining execution is controlled by `swarm_run`. The `swarm_run` goal is defined as

```
swarm_run :-
   iris_open,
   swarm_restart_dataspace,
   init_handler,
   repeat,
   swarm_step,
   !,
   kill_handler,
   iris_close.
```

The goal `iris_open` initiates communication with the Silicon Graphics Personal Iris over the Ethernet. `swarm_restart_dataspace` uses the provided `initialize_dataspace` goal to create the initial dataspace. `init_handler` creates a dialog box with which the user can interact with SwarmExec. After this initialization is completed, the goal `swarm_step` is repeatedly called. `swarm_step` succeeds if the execution of SwarmExec finishes (because the user stops execution) and fails if the SwarmProlog program successfully executed another step; coupled with the `repeat` goal, this means that `swarm_step` will be executed until the SwarmProlog program finishes. Once this occurs, the goals `kill_handler` and `iris_close` terminate the interaction with the user and with the Personal Iris, respectively.

The goal swarm_step performs the operations associated with performing a single atomic action of the underlying Swarm program. This goal is:

```
swarm_step :-
   sw_rsrc_interaction(BUTTON, DoAStep, DoQueries,
DoVisualization, SchedulingPolicy),
   ( DoQueries > 0 -> sw_do_queries(BUTTON,DoAStep) ; true ),
   !,
   (
      BUTTON =:= 5 -> true
   ;
      BUTTON =:= 4 -> swarm_restart_dataspace, !,fail
   ;
      DoAStep > 0 ->
      (
         sw_select_trans(SchedulingPolicy,Trans)
         ->
         sw_group(Trans,Group),
         sw_readyupdate,
         sw_execgroup(Group),
         sw_doupdate
         ;
         true
      ),
      ( DoVisualization > 0 -> sw_do_visualization ; true ),
      !,
      fail
   ;
      fail
   ).
```

The first action taken in the step is to interact with the user through the previously-mentioned dialog box; this is accomplished by the sw_rsrc_interaction goal. The dialog box, depicted in Figure 1, has five buttons to control the progress of SwarmExec and three "radio buttons" which control other aspects of behavior. If any of the five buttons is selected by the user, the button number is returned through the variable BUTTON; otherwise the value 0 is returned. The three radio buttons control the values returned in DoQueries, DoVisualization, and SchedulingPolicy. Finally, the value of DoAStep is positive if SwarmExec should perform one atomic action of the SwarmProlog program.

The rest of the goal simply consists of properly handling the results of the user interaction. If DoQueries is true, the goal sw_do_queries is performed (at this time, this goal has no effect; it is included for a designed enhancement in which the user will be able to have continuously-updated queries over the dataspace displayed in separate Macintosh windows). If the user selects button 5 (Quit) the goal terminates successfully, resulting in termination of the main swarm_run goal as described previously. If the user selects button 4 (Restart) the dataspace is re-initialized and control returns to swarm_run.

If a step should be performed, a transaction is first selected by the goal sw_select_trans; if a transaction is found, the four steps from sw_group to sw_doupdate execute the group and update the dataspace as described in the next subsection. Finally, if DoVisualization is true the visualization mappings are applied and the results sent to the Personal Iris. Note that swarm_step succeeds only when the user selects the Quit button.

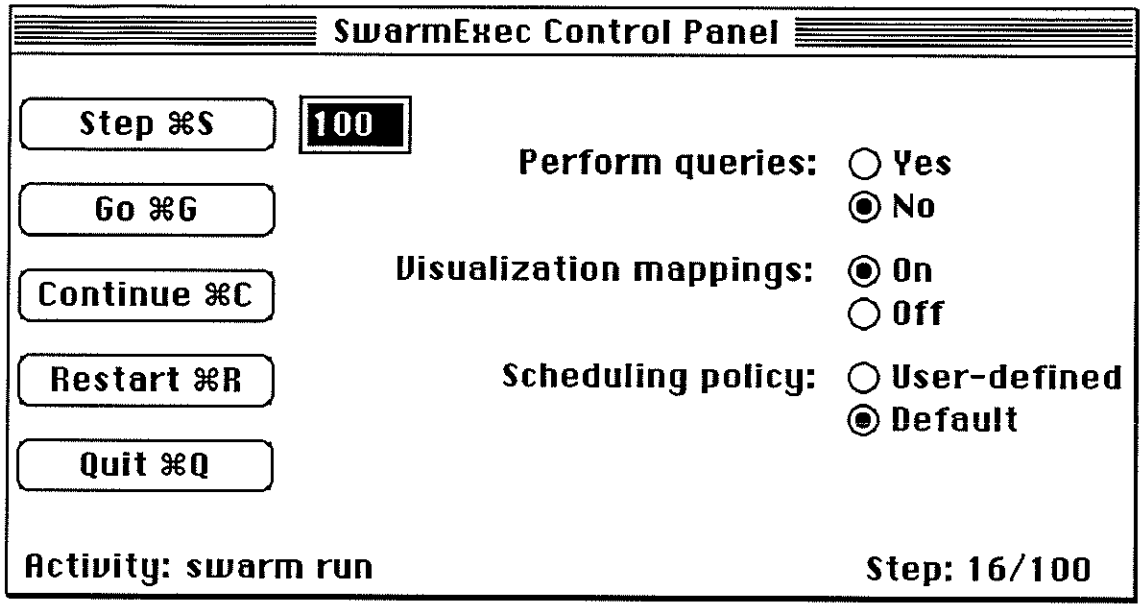


Figure 1. The SwarmExec control panel.

3.2 Selection and Execution of Synchronic Groups

Selection of a transaction is performed by the `sw_select_trans` goal. Currently, this goal merely locates a transaction in the dataspace. It is included to permit later expansions in which the user can exercise a greater degree of control over the selection and sequencing of transaction executions.

Once a transaction is selected the `sw_group` goal determines the synchronic group to which the transaction belongs. `sw_group` simply performs a exhaustive search from the initial transaction through the closure of the synchronic relation, using standard search techniques. The synchronic group is represented as a Prolog list of transactions with parameters. Note that the group may contain transactions which are not actually present in the dataspace; by the Swarm definition, only those transactions which are actually present will be executed.

The next step is to prepare for execution of the group. The `sw_readyupdate` goal performs this task. The most important function of this goal is to initialize a collection of counters which are used in the processing of the special Swarm queries such as **NOR**, **NAND**, and so forth. These three counters are `sw_total_count`, which counts the total number of local subtransactions (those not involving special queries) and `sw_succ_count` and `sw_fail_count`, which count the number of local subtransactions which succeeded and failed.

Execution of the group then follows, with the `sw_execgroup` goal. This goal simply traverses the list of transactions making up the group. If a transaction in the group is also present in the dataspace, it is executed by the `sw_exectrans` goal:

```
sw_exectrans( Trans ) :-
   assert( sw_deletion(true,trans(Trans)) ),
   call(Trans).
```

The use of the `sw_deletion` tuple here is notable. Swarm's semantics requires that all queries involved in a single atomic operation must occur before the dataspace is altered. We therefore record all dataspace modifications, such as the deletion of a tuple when it is executed, in the Prolog database using the `sw_deletion` and `sw_insertion` tuples. These tuples are then processed after execution of the group to perform the dataspace update. The second component of a `sw_deletion` or `sw_insertion` tuple is

the element to be removed or added; the first component is a tag which gives the type of subtransaction which resulted in the modification (i.e., `local`, `true`, `nor`, etc.).

After the fact that the transaction should be removed from the dataspace is recorded, the transaction is executed using the `call` goal. At this point the syntactic elements discussed in section 2.2 are applied. Recall that the transaction consists of a sequence of subtransactions, each represented in the form

```
setquerytype( type ),
    (   translation of query,
        !,
        translation of action,
        itsucceeded
    ;
        itfailed
    )
```

The `setquerytype` goal makes a temporary record of the type of the subtransaction; this is then referenced when generating `sw_deletion` or `sw_insertion` tuples to determine the first component. The query is performed; if it succeeds, the action is performed (possibly generating `sw_deletion` and `sw_insertion` tuples) and the goal `itsucceeded` is executed. If the subtransaction was a local type, `itsucceeded` increments the counters `sw_total_count` and `sw_succ_count`, thus recording that one subtransaction succeeded. `itfailed` acts similarly but increments `sw_total_count` and `sw_fail_count`.

After all transactions in the synchronic group are executed, the dataspace is updated by the `sw_doupdate` goal. The first action of this goal is to determine what types of special transactions are successful using the `sw_legaltags` goal. This goal examines the counters `sw_total_count`, `sw_succ_count`, and `sw_fail_count` to determine which such transactions are successful. For example, the **OR** transactions succeed if any of the local subtransactions succeeded, i.e. if the count held by `sw_succ_count` is greater than zero. The success of **AND**, **NAND**, and **NOR** special subtransactions is similarly determined. The local subtransactions and those of special type **TRUE** always succeed. If a transaction type is successful, a fact of the form `sw_legaltag(type)` is asserted.

The `sw_deletion` and `sw_insertion` tuples are then processed, with the deletions performed before the insertions (again, this is Swarm semantics). Only tuples with successful tags, as determined by the `sw_legaltags` goal, are processed; the others are discarded. The actual updating of the dataspace is then a matter of traversing the lists of tuples to be deleted and added, with the only slight complication being the need to maintain the spaces as sets of tuples (i.e., no duplicate tuples are permitted).

3.3 Generators and Similar Constructs

The Swarm generator construct has the form

[variables : predicate : list]

and is interpreted in the following manner: For each instantiation of the variables such that the predicate is true, create all the objects in the list. The generator is particularly useful in the initialization part of Swarm programs. As mentioned previously, the behavior of the visualization rules is quite similar. Two additional Swarm constructs, the subtransaction generator (which is used in transactions to specify a collection of subtransactions) and the universal quantifier ($\forall$ v : p : q) are also similar in behavior. In all four cases, we want to determine all instantiations of a collection of variables such that a particular predicate (or query) is true, then perform some action for all such instantiations.

Two Prolog goals, `forall` and `forall_worker`, each with two clauses, suffice to provide all four of these constructs:

```
forall(VarList,Predicate,Action) :-
   bagof(VarList,Predicate,TheBag),
   !,
   forall_worker(TheBag,Action).
forall(_,_,_).

forall_worker([],_).
forall_worker([H|T],Action) :-
   apply(Action,H),
   forall_worker(T,Action).
```

`forall` uses the built-in Prolog goal `bagof` to find all instantiations of `VarList` such that `Predicate` can be satisfied. The list of all such instantiations is bound to `TheBag`, which is then passed to `forall_worker`. The latter goal recursively iterates the list and performs the action for each variable instantiation in the list. Although `forall` is sufficient, we include goals named `sw_generator`, `sw_subtrans_generator`, and `sw_visualization_rule` for clarity; all are identical and interchangeable.

As a general rule, each invocation of `forall` will be accompanied by definitions of two additional goals used for the `Predicate` and `Action` goals. Because of the behavior of `bagof`, some extra syntactic elements must be included in invocations of `forall`. These elements serve to existentially quantify the variables of `VarList` for the `bagof` goal, ensuring all possible instantiations will be found. For example, the query

(∀ x, y : tuple1(x,y) : tuple2(x,y,3))

would be translated into SwarmProlog as

```
forall([X,Y], X^Y^goal1(X,Y),goal2)
```

with the subsidiary goals `goal1` and `goal2` being defined as

```
goal1(X,Y) :- tuple(tuple1(X,Y)).
goal2(X,Y) :- tuple(tuple2(X,Y,3)).
```

Note that this goal will succeed if and only if for every instantiation of `X` and `Y` such that `goal1` is true, `goal2` is also true. This is typical of a universal quantifier; in the case of a generator or visualization rule, `goal2` would not be a predicate but would instead be some action (probably an assertion) which always succeeds, and the `forall` would then also always succeed (as desired of a generator).

A similar `thereexists` goal is also provided, with the same syntactic form (which is not actually required in this case; however, keeping the forms similar simplifies the programmer's task). `thereexists` succeeds if it can find any instantiations of the variables for which both predicates succeed:

```
thereexists(V,P,Q) :- P, apply(Q,V).
```

Finally, several goals are provided to produce sums, products, and so forth. These are all similar in form and closely related to the generator, as illustrated by `sw_sum` and `sw_sum_worker`:

```
sw_sum(Goal,Expr,Sum) :-
   bagof(Expr,Goal,TheBag),
   sw_sum_worker(TheBag,Sum).
sw_sum(_,_,0).

sw_sum_worker([],0).
sw_sum_worker([H|T],Sum) :-
   sw_sum_worker(T,SubSum),
   Sum is SubSum + (H).
```

`sw_sum` collects all instantiations of the `Expr` such that the `Goal` is true and passes the resulting list to `sw_sum_worker`, which recursively sums the expressions. The result is "returned" through the variable `Sum`.

3.4 Visualization Mappings

The goal `sw_do_visualization` applies the visualization rules to the current Swarm dataspace and sends the results to the Personal Iris. `sw_do_visualization` first copies the current proof space and object space to the previous proof and previous object spaces respectively, then clears the proof, object, and animation spaces. Each of the three mappings – proof, object, and animation – is then applied in turn to generate the spaces. Finally, the animation space is sent to the Personal Iris over the interface described in section 4.2.

The key operation in this sequence is the application of each mapping. As discussed in section 2.3, the mapping is represented by a list of names of rules which are to be applied; definitions of the rules are given separately using the `sw_visualization_rule` goal described in the previous section. Applying rules is simply a matter of finding the list of names, then applying the `vis_apply_rules` goal which recursively calls each rule in the list:

```
vis_apply_rules([]).
vis_apply_rules([H|T]) :- call(H), vis_apply_rules(T).
```

4 Interfaces

SwarmExec has two primary interfaces through which it interacts with the user and the Personal Iris graphical engine. Both of these interfaces are implemented as "CODE resources", a Macintosh term referring to a pre-compiled code segment which can be loaded and used by other applications. AAIS Prolog provides several goals which allow easy use of such resources.

4.1 User Interface

The user interface has been briefly described in section 3.1. When the CODE resource is initialized by the goal `init_handler`, the dialog box depicted in Figure 1 is created. The goal `sw_rsrc_interaction` is used to interact with the CODE resource. `kill_handler` disposes of the dialog box when interaction is complete.

The five buttons to the left of the dialog box control the progress of the SwarmExec interpretation of the SwarmProlog program. The first button, Step, has an associated "edit box" (displaying 100 in Figure 1). If the user enters a number greater than 0 into the edit box and hits Step, SwarmExec will perform steps of the SwarmProlog program until the number entered in the edit box is reached; execution then pauses. As the steps are performed, the step number is displayed in the lower right corner of the dialog box. The second button, Go, is similar to Step except execution continues without limit.

The Continue/Pause button allows execution (either in Step or in Go) to be temporarily halted and then resumed. The fourth button, Restart, starts the SwarmProlog computation from the beginning by retracting the current dataspace and re-initializing it using the user-provided `initialize_dataspace` routine. The final button, Quit, ends SwarmExec.

On the right side of the dialog box are three pairs of "radio buttons"; in each pair, exactly one of the two choices is selected at any time. These three pairs influence the behavior of SwarmExec on each step; the first determines whether query windows should be updated, the second whether visualization should be performed, and the third whether a user-defined scheduling policy should be used. (At this time, only the second button has any effect; the others are included for planned enhancements.)

4.2 Graphical Engine Interface

The Macintosh and Personal Iris communicate over an Ethernet connection. The Personal Iris runs IRIX®, a UNIX®-like system which implements the standard UNIX socket interfaces. The graphical engine interface on the Macintosh provides UNIX-compatible communications between the machines using a client-server protocol in which the Personal Iris acts as server.

The interface is implemented as a CODE resource. The goal `iris_open` initiates communication with the Personal Iris server daemon. The daemon initiates a process on the Iris which opens a graphical window, reads and interprets the animation space tuples sent by the Macintosh, and generates the images. The goal `iris_close` ends the connection, resulting in termination of the graphical process on the Iris. The operation of the Iris graphical process, including a description of the animation space tuples and interpretation, is described in a companion paper.

The required communication is bidirectional. After each atomic action of the SwarmProlog program, SwarmExec sends the the animation space tuples to the Personal Iris. When the Personal Iris has finished displaying the graphics associated with an animation space, it sends back an acknowledgment; this protocol assures that the machines remain coordinated.

Tuples are sent from the Macintosh to the Iris in the form of strings. Each animation space tuple is converted into a string and transmitted using the goal `iris_send`. For efficiency, the CODE resource buffers the strings; the goal `iris_flush` is used to force transmission of the buffer to the Iris. After transmission of the space is completed, SwarmExec pauses until it receives the acknowledgment from the Personal Iris.

5 Acknowledgments

The authors would like to thank Dr. Jerome R. Cox of the Department of Computer Science at Washington University for his support. This research was supported in part by the National Fellowship Program in Parallel Processing, supported by DARPA/NASA and administered by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

Bibliography

Research on visualization of concurrent computations at Washington University:

Cox, K. C., *Visualization of Concurrent Computations* (Doctor of Science Dissertation Proposal), Technical Report WUCS-89-32, Department of Computer Science, Washington University in St. Louis (June, 1989).

Cox, K. C. and Roman, G.-C., "Visualizing Concurrent Computations", Technical Report WUCS-90-31, Department of Computer Science, Washington University in St. Louis, September 1990. Submitted to the *13th International Conference on Software Engineering*.

Cox, K. C., "Visualization in Concurrent Contexts: A Model", Technical Report WUCS-91-7, Department of Computer Science, Washington University in St. Louis, November 1990.

Cox, K. C., Wilcox, C. D., and Plun, J. Y., "SwarmExec: A Prolog-Based Execution Engine for a Shared-Dataspace Language with Visualization Capabilities", Technical Report WUCS-91-8, Department of Computer Science, Washington University in St. Louis, December 1990.

Cox, K. C., "SwarmView: A Graphical Engine for the Interpretation and Display of Visualizations", Technical Report WUCS-91-9, Department of Computer Science, Washington University in St. Louis, January 1991.

Cox, K. C., "SwarmView Animation Vocabulary and Interpretation", Technical Report WUCS-91-10, Department of Computer Science, Washington University in St. Louis, November 1990.

Roman, G.-C. and Cox, K., "A Declarative Approach to Visualizing Concurrent Computations", *IEEE Computer*, Vol 22 No. 10, pp. 25-36 (October 1989).

Roman, G.-C. and Cox, K., "Declarative Visualization in the Shared Dataspace Paradigm", *Proceedings of the 11th International Conference on Software Engineering* (May 1989).

Swarm notation and proof system:

Cunningham, H. C., *The Shared Dataspace Approach to Concurrent Computation: The Swarm Programming Model, Notation, and Logic*, Doctor of Science Dissertation, Department of Computer Science, Washington University in St. Louis, August, 1989.

Cunningham, H. C. and Roman, G.-C., "A UNITY-Style Programming Logic for a Shared Dataspace Language," *IEEE Transactions on Distributed and Parallel Computing* Vol.1, No. 3, pp. 365-376 (July 1990).

Roman, G.-C. and Cunningham, H. C., "A Shared Dataspace Model of Concurrency - Language and Programming Implications," *Proceedings of the 9th International Conference on Distributed Computing Systems*, pp. 270-279 (June 1989).

Appendix 1. Grammar

The following is a BNF grammar of the language. Terminals are in **bold**. The symbol λ represents the null string. The "terminal" **identifier** represents any legal identifier (anything described by the LEX-style regular expression [a-z][a-zA-Z_0-9]*); **number** represent any legal numeral, either integer or real.

language	::=	transition_list
transition_list	::=	transition transition_list
	|	λ
transition	::=	pge ; transition
	|	**end;**
pge	::=	type_name (attribute_list)
	|	type_name ()
type_name	::=	**identifier**
attribute_list	::=	attribute , attribute_list
	|	attribute
attribute	::=	attribute_name = expression
attribute_name	::=	**identifier**
expression	::=	function
	|	constant
function	::=	primitive_function
	|	[primitive_function_list]
primitive_function_list	::=	primitive_function , primitive_function_list
	|	primitive_function
primitive_function	::=	**identifier** (constant_list)
constant_list	::=	constant , constant_list
	|	constant
constant	::=	**number**
	|	**t_max**
	|	[constant_list]

Appendix 2. Graphical Objects

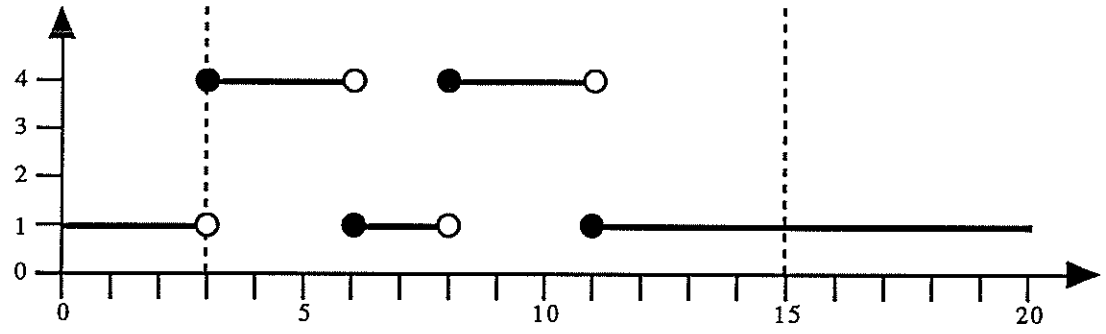
The following graphical objects are provided by the interpreter. All objects have a *lifetime* attribute, which is of type list of two numbers. The type *coordinate* is a shorthand for "list of three numbers" and specifies the X/Y/Z coordinates of the point. The type *color* is "list of three numbers" 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	Notes
point	position	coordinate	[0, 0, 0]	location of point
	color	color	white	color of point
line	from	coordinate	[0, 0, 0]	one endpoint
	to	coordinate	[0, 0, 0]	the other endpoint
	color	color	white	
	width	number	1	width of line (screen pixels)
rectangle	corner	coordinate	[0, 0, 0]	lower left corner
	xsize	number	1	dimensions
	ysize	number	1	
	color	color	white	
	fill	number	0	if non-zero, rectangle is filled
	xrot	number	0	rotation about X-axis, degrees
	yrot	number	0	rotation about Y-axis
	zrot	number	0	rotation about Z-axis
crect	center	coordinate	[0, 0, 0]	centroid
	xsize	number	1	dimensions
	ysize	number	1	
	color	color	white	
	fill	number	0	if non-zero, rectangle is filled
	xrot	number	0	rotation about X-axis, degrees
	yrot	number	0	rotation about Y-axis
	zrot	number	0	rotation about Z-axis
polygon	vertices	list of coordinates	[[0, 0, 0]]	in the order to be connected
	color	color	white	
	fill	number	0	
circle	center	coordinate	[0, 0, 0]	
	radius	number	1	
	color	color	white	
	fill	number	0	
	xrot	number	0	as in **crect**
	yrot	number	0	
	zrot	number	0	
sphere	center	coordinate	[0, 0, 0]	
	radius	number	1	
	color	color	white	

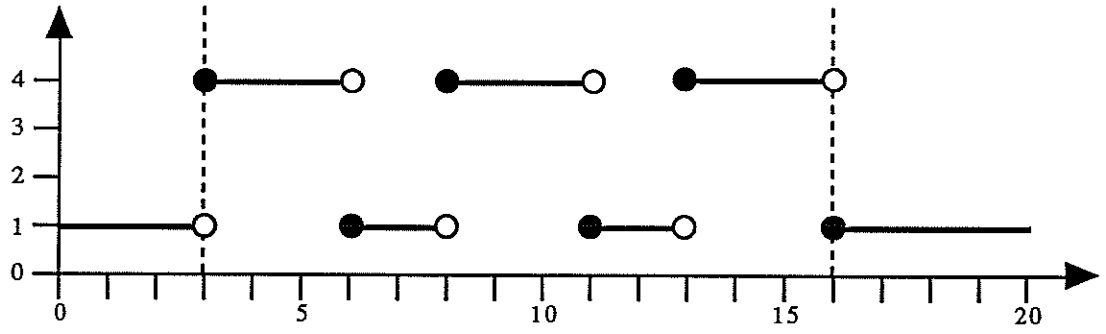
Appendix 3. Functions

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. Ass 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 following diagram gives some examples of this for clarification. Both graphs show a square wave with $p_{on} = 3$ and $p_{off} = 2$. In the upper 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 lower graph there is sufficient time for an addition v_{on} period to be included.



square(3,15, 3,2, 4,1)



square(3,16, 3,2, 4,1)