COMPOUND MODULES AS GOALS

UDK 681.3.06 PROLOG:519.682

Monika Kapus-Kolar
IJS, Ljubljana

Osnovna objektna interpretacija jezikov tipa Prolog je kreiranje in brisanje modulov. Nekateri jeziki te družine (npr. Delta Prolog) uvajajo vzporedno/zaporedno kompozicijo in eksplicitno komunikacijo, zaradi česar so primerni za specifikacijo komunikacijskih protokolov. Najvažnejša ideja pričujočega dela je ločitev pojma modula od pojma cilja, tako da lahko vsak modul sodeluje v več ciljih (sestavljenih modulih). Jezik Hornovih stavkov z vzporedno/zaporedno kompozicijo atomičnih dogodkovnih in nedogodkovnih ciljev smo dopolnili z novimi sintaktičnimi elementi, ki v luči nove izvedbene semantike omogočajo bolj jedrnato specifikacijo komunikacijskih protokolov, posebno tistih, pri katerih so potrebne številne operacije na posamezni zapleteni podatkovni strukturi. Jedrnatost dosežemo s temeljito izrabo možnosti, ki jih nudi struktura sestavljenih modulov, ki je ponavadi precej preprostejša kot sintaksa atomičnih modulov.

The basic object paradigm of Prolog-type languages is creation and deletion of modules. Some languages of the family (e.g. Delta Prolog) introduce parallel/sequential composition and explicit communication, what makes them suitable for specification of communication protocols. The main contribution of the paper is separation of the "module" concept from the concept of "goal", so that a module may participate in several goals (compound modules). Some syntactic enhancements have been proposed for the language of Horn clauses with parallel/sequential composition of atomic event and non-event goals, which in the light of the new operational semantics provide for more concise specification of protocols, particularly those requiring multiple operations on a complicated piece of data. The core idea of the new language is full exploitation of the structure of compound modules, which is usually much more simple than the syntax of atomic modules.

1. Introduction

The basic idea behind languages of the Prolog family is to find solutions to a problem (goal) by its reduction into more and more trivial subproblems (subgoals).

A Prolog program, [4], is a set of universally quantified first order axioms (Horn clauses) of the form

$A :- B_1, B_2, \ldots, B_n$

where the A and the B's are atomic formulae, also called atomic goals. A is called the clause's head and the B's are called its body. The computation proceeds by selection of a goal A_i from the current conjunctive goal $(A_1, A_2, \ldots, A_m)$, which is then reduced with a selected clause

$A' :- B_1, B_2, \ldots, B_k$

where A and A' must be unifiable via the most common substitution θ. The reduction step transforms the current goal into

$(A_1, \ldots, A_{i-1}, B_1, \ldots, B_k, A_{i+1}, \ldots, A_m)\theta$.

In the process of unification, some of the variables of the initial goal are assigned values, which constitute the output of the computation. The computation terminates successfully, when the initial goal is reduced into an empty goal, but may terminate unsuccessfully, if no further reduction is possible, or not terminate at all. Several successful computations of a program may exist, resulting from various selections of clauses for reduction.

Graphical representation of a Prolog program is an AND/OR tree with AND nodes modelling composition of goals into a clause body and OR nodes enumerating the suitable clauses for reduction of a particular goal. Several goals can be reduced in parallel (AND-parallelism) and several alternative solutions searched for in parallel (OR-parallelism). True concurrency is allowed, if atomicity of reduction steps is preserved.

The inherent AND-parallelism of Prolog programs makes them suitable for operational specification of communication protocols. An AND subtree of the AND/OR tree models a particular execution of a system, while the search of the entire AND/OR tree represents verification of all possible behaviours of the system. Note, that logic programming is also suitable for axiomatic specification and verification of communication protocols, but the paper does not deal with this aspect.

The language has two major deficiencies: First, by introducing AND-parallelism, the execution order is controlled by the goal-subgoal relation only, so it is difficult to describe sequential protocols. Second, communication between goals is implicit and asynchronous via common variables, while partners in communication protocols are usually loosely coupled (not sharing any variables).

To solve the first problem, many Prolog-type languages (e.g. Delta-Prolog (DP) [6,7], Concurrent Prolog (CP) [5], M-Prolog [9]) make distinction between the parallel (¦¦) and the sequential (¦) composition of goals. Declaratively, the two composition operators are equivalent to the AND-operator, but serve to specify the execution order in the spirit of CCS [1], CSP [2] or LOTOS [3].

In the literature, we meet two types of explicit communication: communication on the level of variables and communication on the level of atomic goals. An example of the first type are the "read-only" variables in CP, modelling asynchronous broadcast. An example of the second type are events in DP. Here, the willingness of a module to participate in an event of a particular type is expressed by a goal of a special kind - an event goal, which is successfully reduced in cooperation with some peer event goals of the (other) modules. The DP concept of events allows various cooperation schemes, differing in the number of participating modules, the degree of their synchronization and in side-effects of a particular event, while in CP, a single cooperation scheme is defined. Aiming towards an event-order specification language, the DP concept of events is adopted in the paper and extended.

2. The Architectural Aspect of the Language

An instantaneous representation of a system, described by a set of Horn clauses, is a tree - the **architectural tree**. Nodes of the tree are hierarchical parallel/sequential compositions (trees) of goals. The root represents the top-level structure of the system, i.e. its static architectural components (the initial execution goal). Each atomic goal represents a declaration of a particular module of the system. When an atomic goal is reduced with a clause, the body of the clause is introduced in the tree as a descendant of the goal. A subtree, attached to an atomic goal, represents dynamic architecture of the module, declared by the goal. After a node has been successfully executed (all its atomic goals reduced to TRUE), it is deleted from the tree.

As the overall activity of a system is represented as creation and deletion of modules, there is no evident distinction between a module, representing a quasi-static architectural component of the system, and a module, representing a procedure. Such semantic distinction belongs to a lower level of abstraction. The only feature that matters is the capability of the language to specify loosely coupled modules. Modules are declared "loosely coupled" simply by keeping their variable-sets disjoint.

Atomic goals, which are event goals, do not generate subtrees, but are reduced in events. Looking at an event as a common action of several modules, it is a free module, not embedded in the architectural tree, with its submodules residing in various nodes of the tree. According to the previous paragraph, it is difficult to say which module does a particular event goal belong to, but it is no doubt that it belongs to a certain node. As events should serve for cooperation between loosely coupled modules, it is reasonable to declare that event goals, participating in a particular event, must not belong to the same node.

The word "goal" should not be used in the architectural context - it denotes the intention of the current architecture to change, but at that point, we are only interested in an instantaneous picture of a system. Therefore, atomic goals are rather called **atomic modules**. They are basic elements of nodes and are usually combined into **compound modules** by parallel/sequential composition operators.

Modules are further classified into **explicit** and **implicit** ones. They can be best identified by observing a clause body (e.g. Fig.1):

```
compound module: (A¦B¦(C¦¦D¦¦(E¦F)))

explicit modules:

A, B, C, D, E, F
(E¦F)
(C¦¦D¦¦(E¦F))
(A¦B¦(C¦¦D¦¦(E¦F)))

implicit modules:

(C¦¦D), (D¦¦(E¦F)), (C¦¦(E¦F))
(A¦B), (B¦(C¦¦D¦¦(E¦F)))
```

Fig.1: Explicit and implicit modules of a compound module.

Atomic modules and bodies are explicit modules. Operands of parallel and sequential composition operators are explicit modules. Any proper subset of modules with more than one element, belonging to a parallel composition of modules, is an implicit module and itself a parallel composition of modules. Any proper subsequence of modules with more than one element, belonging to a sequential composition of modules, is an implicit module and itself a sequential composition of modules.

The above definition illustrates the **module-grouping and module-ordering role of** the two **composition operators**. The parallel composition operator groups modules into sets and the sequential composition operator groups modules into sequences. Explicit modules are the actual and implicit modules the potential groups of modules.

3. The Operational Aspects of the Language

3.1. Compound Modules as Goals

If a module is declared to be a **goal**, it specifies a pending action. The goal becomes "executed", when the action is no longer pending. A node gains the right to be deleted from the architectural tree, after all its goals have been executed.

In DP, only atomic modules are goals, every atomic module is a goal and the pending action of each goal is its reduction into TRUE. The main contribution of the paper is the idea, that **the concept of module should be strictly separated from the concept of goal, so that a module might participate in several goals (compound modules)**. Note the importance of the word "participate" - a module itself is not necessary a goal, but every module should be included into some goal, otherwise it has no practical role in the system.

Motivation for the new idea has been the fact that in most cases events serve just for some kind of unification of modules, belonging to various nodes. From this aspect, event goals are just communicated pieces of data. Assuming that there is a group (composition) of modules, there might be several nodes, each interested in a particular subgroup of the group and willing to observe the whole subgroup in a single event. It is much more elegant to enumerate members of the group once and to declare that each subgroup of the group is an event goal, than to enumerate all subgroups as atomic event goals. If a group of modules is a set (parallel composition), it might serve as a data-base (see section 5 for the example in Fig.6); if it is a sequence (sequential composition), it might serve for specification of a data-stream with multiple observers, each waiting for a particular subsequence. If the unifying event goals are parallel compositions of modules, the unification rule could be less strict (unification possibly preceded by permutation of modules) than for event goals, which are sequences. The fact is, that sequences and sets already exist in clause bodies, so why not to use them (section 5). Similarly, if there has been a non-event goal (a procedure) executed, why should its declaration part be duplicated just to communicate its exit results to an observer.

The central operational concepts of our language are **reduction goals** and **event goals**. The two names indicate that the concept of reduction has been separated from the concept of event - without this step the realization of our ideas would not be possible. The role of reduction goals is twofold: First, if they are atomic, they enforce execution of procedures in the ordinary sense (like non-event goals in DP). Second, they control execution order by imposing hierachical execution of goals: No goal can be executed, if some of its submodules are pending reduction goals, so that no event can occur on exit results of a procedure, until they are available. Because of the nature of the two roles, only explicit modules may be reduction goals, so that a reduction goal must be entirely contained in another reduction goal, or not at all.

Event goals support the idea of overlapping goals, as they may be explicit or implicit modules and may partially overlap. An event goal is executed by an event, where the participating part of a node is exactly the module, representing the goal. Event goals impose no particular execution order - a dangerous dimension of freedom, which is partially compensated with the execution-ordering role of reduction goals and composition operators.

Depending on its reduction type (see section 3.3.), execution of a reduction goal is independent from events or a reduction goal is a submodule of an event goal and executed simultaneously with the event goal.

In DP, event goals never generate descendants in the architectural tree, while event goals with atomic submodules, which are reduction goals, requiring the ordinary type of reduction, do. Nevertheless, when such an event goal gains permission for execution, all such reduction goals within it have already been executed and their descendants deleted from the tree.

3.2. The Execution-ordering and the Selection Role of Composition Operators

The basic role of composition operators is their module-grouping and module-ordering role, but for easy specification of sequential protocols they must also be assigned the **execution-ordering role**. To employ the full power of the two roles, it must be possible to use them independently, while in DP they are not separated. Therefore we define that the parallel and the sequential composition operator have no a priori execution-ordering role.

When considering the execution-ordering role of composition operators, the distinction between compound modules, which are sets, and those, which are sequences, is irrelevant, because the attribute parallel/sequential, attached to composition operators, applies to another role. Therefore, all compound modules will be treated as sequences of modules.

A composition operator separates a sequence of modules into the left and the right subsequence. A DP sequential composition operator forces the goals from the left subsequence to be executed before the goals from the right subsequence. But if a goal extends to the left and to the right subsequence, which subsequence does it belong to? Another question: Should the composition operator delay actions on the right subsequence until the reduction goals from the left subsequence have been executed or until the event goals have been executed or, perhaps, just until all goals from the left subsequence have been created. The answer depends on the nature of a particular application, but as default we propose that creation of all goals of a node is an atomic action and that execution of a goal is treated as atomic in the sense that if a goal A must be executed before a goal B, then (at least virtually) the execution of the goal B may not even start before the execution of the goal B has been completed.

The execution order is effected by reduction goals, implying that some goals must be executed before some others. Beside that, some reduction goals are executed simultaneously with an event goal, implying an OR composition of relations, specifying simultaneous execution. In addition, the language should facilitate specification of further relations of the "before" type.

Such a specification should be concise, therefore we propose that goals are referred to simply by their position in the node and that the composition operator, to which a particular "before" relation is attached, is carefully selected such that the goals of the relation are easily specified relatively to the position of the operator (section 5).

The last requirement suggests distribution of such relations all over the node. Anyway, the execution order is determined by considering all the relations simultaneously (in DP, it is sufficient to consider relations, implied by sequential composition operators, in a particular order). If a set of relations is in contradiction (i.e. $(a<b)$ and $(b<a)$), they are ignored. With that rule, a programmer is free to specify any relation, and if possible, it will be respected (e.g. Fig.2).

In comparison to LOTOS, the language lacks two important features: a construct for expressing intelligent selection (guarded commands) and a construct for expressing disruption of processes. If guards are not located at the meta level (as, for example, in Two-level Prolog [8]), but at the object level, both problems have a simple common solution - **exclusive composition operators**.

```
body: (a¦{ULS<RS}
       b¦{RS.A<URS.ULS}
       c¦{LS.A<URS}
       d)#M: (event goal - true;
              reduction state - no-reduction)#

execution-ordering relations:

1.composition operator:
  {(a), (a¦b), (a¦b¦c), (a¦b¦c¦d)} <
  {(b), (c), (d), (b¦c), (c¦d), (b¦c¦d)}

2.composition operator:
  {(c), (d)} < {(b¦c), (a¦b¦c), (b¦c¦d),
                (a¦b¦c¦d)}

3.composition operator:
  {(a), (b), (c)} < {(d), (c¦d), (b¦c¦d),
                     (a¦b¦c¦d)}

ignored relations:
  {(b)<>(a¦b¦c¦d),
   (c)<>(a¦b¦c), (c)<>(a¦b¦c¦d),
   (d)<>(a¦b¦c), (d)<>(a¦b¦c¦d)}
```

Fig.2: The execution-ordering role of composition operators.

A compound module may be a non-exclusive or an exclusive composition of modules. The non-exclusive forms of the sequential and the parallel composition operator are ¦ and ¦¦, while their exclusive forms are / and //, respectively. Each module of an exclusive composition of modules is attached a set of goals (drawn from the set of all goals of the node) representing the guard of the module (see section 5 and Fig.3 for an example).

```
body: (((a#M: (reduction state - pending;
               reduction type - weak-common)#
         ¦b#M: event goal - true#
         )
        ¦c
        ¦d#M: event goal - true#
       )[(11)M]
      //
        (e#M: (reduction state - pending;
               reduction type - weak-common)#
        ¦f
        ¦g
        ¦h#M: event goal - true#
        )
      //
        (i#M: (reduction state - pending;
               reduction type - weak-common)#
        ¦j#M: event goal - true#
        )
      )#M: reduction state - no-reduction;
        WM: event goal - true#

scenario:
  execution of event goal
     (((a¦b)¦c¦d)//(e¦f¦g¦h)//(i¦j))
  -> ((a¦b)¦c¦d) not ready for selection,
     (e¦f¦g¦h) and (i¦j) ready for selection
  -> the node transformed into (e¦f¦g¦h)
     or into (i¦j)
  -> execution of event goal (h)
     (or event goal (j))
```

Fig.3: An example of a guarded command.

After the guard of a module has been executed, the next operation on the module may only be its selection for further execution or its deletion from the system. Several modules with executed guards may exist, but exactly one of them is selected non-deterministically and the entire exclusive composition of modules replaced by the selected module, which from that moment behaves like an ordinary module. Because guards are regular processes of a system and one of the modules in exclusive composition disrupts the others, this is a model of process disruption, more general than the one of LOTOS.

3.3. Execution of Reduction Goals

In DP, a reduction goal is executed either by the ordinary type of reduction (reduction of a goal into more and more trivial subgoals), or by participation in an event as exactly the whole event goal. In our language, a reduction goal can also be executed in event, where it is just a submodule of the relevant event goal. Hence, according to this criterion, there are three types of reduction.

1. **On-the-spot reduction** is the ordinary type of reduction and could serve for specifying an internal process of a node. It may be **non-trivial** or **trivial**. Reduction of an atomic module is non-trivial, as it potentially creates descendants in the architectural tree. Reduction of a compound module is trivial, as it is just an observation that all the reduction goals within the module have been executed.

2. **Strong common reduction** takes place, when a module is a reduction goal and an event goal at the same time (the DP-type event goal). When the event goal is executed, the reduction goal is executed, too. The adjective "common" steams from the fact that the participants can execute the reduction as a common action, without any of them executing the reduction on the spot. Common reduction could serve for exchange of values of any origin.

3. **Weak common reduction** is like strong common reduction, but the reduction goal may also be just a submodule of the relevant event goal.

To facilitate observation of final (exit) results of modules, we introduce a fourth type of reduction, which is a weak common reduction with some additional requirements:

4. **Observed reduction** takes place, when a reduction goal participates in an event, in which one of the participating event goals contains a submodule, unifiable with the reduction goal. That submodule must be an executed reduction goal with reduction type "on-the-spot" or must have been unified with such a goal in one of the previous events. In this way, the node does not have to execute the reduction goal on the spot (that might be a difficult operation, if the event goal is atomic), but just gathers the necessary results by observing someone, who has already executed a matching goal on the spot or has learned the results from another node. Observation of exit results is important from several aspect: First, it can strongly reduce the execution effort. Second, it can reduce non-determinism by forcing various nodes to accept the same solution to various incarnations of the same problem. Third, it could serve to implement monitors of overall system activity. Fourth, the concept of exit results is another step towards LOTOS.

On-the-spot reduction may be thought of as being executed in the node, containing the reduction goal, observed reduction as executed in another node and common reduction as executed in the space between nodes. Observed and common reduction are treated as trivial, as they do not create descendants.

At the time of its creation, each module is assigned its reduction type and state. The possible reduction types are the four types, declared above. The possible states are "executed", "pending" and "no-reduction".

If a module is a reduction goal, its initial reduction state is "pending" and its initial reduction type may be any type. If a module is not a reduction goal, its initial reduction state is "no-reduction" and its reduction type is "weak-common". When a reduction goal is executed, its reduction state is set to "executed".

If reduction type of a reduction goal is *"strong-common"*, *the module is an event goal by* definition. If reduction type of a reduction goal is "observed" or "weak-common" and the module is not a submodule of any event goal, it is an event *goal* by definition.

Implicit modules are never able to propagate exit results of an on-the-spot reduction, while explicit modules are always able to do that. When created, reduction goals with the reduction type "on-the-spot" are assigned unique reduction identifiers. Reduction identifiers of other modules are undefined, until they begin to propagate results of an on-the-spot reduction. In that case, their reduction type and state are set to "on-the-spot" and "executed" and their reduction identifier is set to the identifier of the reduction they propagate. Reduction identifiers have been introduced to distinguish among various incarnations of a procedure and to facilitate specification of events with a limited number of active roles, in which the participating event goals may enroll (see section 5 for the example in Fig.7).

Each atomic reduction goal with reduction type "on-the-spot" is attached a predefined or user-written procedure for unification with clauses' heads. Similarly, heads are attached procedures for unification with reduction goals. Such procedures may be treated as sets of constraints. A necessary condition for reduction of a reduction goal with a particular clause is, that their proposed constraints can be satisfied simultaneously. The reduction is executed exactly according to the constraints, proposed by the goal or the head. The default unification procedure is the most common substitution.

3.4. Execution of Event Goals

An event goal can execute, if it meets a group of suitable event goals. Execution must be fair in the sense that an event goal, which is ready to execute, must not wait indefinitely, if suitable groups keep occurring, and must be allowed to join a group, which is able to accept it. If there is more than one suitable group, the event goal selects between them non-deterministically. While an event goal is waiting for execution, its variables might be getting assigned by other goals, so its selectivity improves with time.

Each event goal is attached a predefined or user-written procedure for its execution. A necessary an sufficient condition for a group of event goals to realize an event is, that their proposed constraints can be satisfied simultaneously. The event is executed exactly according to the constraints, proposed by at least one participant.

When a group of event goals is ready to execute a common event, the pending event is created in the system as a free module, which may (or may not) later be executed (deleted from the system). Each of the participating event goals (according to its synchronization requirements) may be executed simultaneously with the event or before the event, but never after the event. Event goals remain formal participants of an event, until the event has been executed, even if they have already been executed or even deleted from the system. If not forbidden by the existing participants, new participants (sometimes even an unlimited number of them) may join with time. In that case, a participant, which has not requested an immediate execution, can not be executed while new participants could potentially change its execution results (*e.g. if they could assign some of its* remaining variables). Regardless the type of the execution procedure, an event must mandatory execute all reduction goals, that are submodules of the participating event goals, and must not introduce any new tight coupling of nodes.

We propose to classify requirements, posed by execution procedures, into those considering

1. the number of participants,

2. timing relations (e.g. synchronization, delays),

3. relation between the state of the participants before and after the event, and

4. potential side-effects.

The proposed defaults are: any non-zero number of participants, full synchronization (all event goals executed simultaneously with the event), no special side-effects and the third relation defined by the following unification procedure:

The procedure first tries to make the participating event goals unifiable via substitution, together with a legal distribution of roles. If it succeeds, the event goals are unified via substitution and their pending reductions executed. The exclusive and the non-exclusive version of composition operators are treated as equivalent.

To make the event goals unifiable via substitution, the procedure may apply to them the following transformations, which are not visible after the event:

- permutation of modules in parallel composition,

- grouping of modules (creation of explicit modules for the time of the execution of the event). The new modules are neither reduction nor event goals and are unable to propagate results of an on-the-spot reduction.

The first transformation type supports the idea of using parallel and sequential composition operators for data ordering and reflects the fact that modules in parallel composition are not ordered. The second transformation type facilitates structured observation of unstructured data.

Usually, an event goal (EG) can be transformed in several ways to meet the requirements. In that case, the selection between the transformations is non-deterministic and the procedure acts as a generator of permutations of modules (e.g. Fig.4).

1.EG: (A¦¦B) - ready to receive data into A and B
2.EG: (a¦¦b) - ready to send data-items a and b

possible transformations before unification and the resulting EG:

	1.EG	2.EG	1.EG after unification
			A B
1.	(A¦¦B)	(a¦¦b)	(a¦¦b)
2.	(A¦¦B)	(b¦¦a)	(b¦¦a)
3.	(B¦¦A)	(a¦¦b)	(b¦¦a)
4.	(B¦¦A)	(b¦¦a)	(a¦¦b)

Fig.4: An event, which does not preserve the order of data.

When the event goals have been made unifiable via substitution, sets of corresponding modules can be identified (e.g. Fig.5). We are interested only in the explicit modules of the event goals (the event goals themselves are treated as explicit modules).

1.EG: (a¦(b¦¦C¦¦d))
2.EG: (a¦(X¦¦c¦¦d))

sets of corresponding modules:

{(a),(a)}; {(b),(X)}; {(C),(c)}; {(d),(d)}
{(b¦¦C¦¦d),(X¦¦C¦¦d)};
{(a¦(b¦¦C¦¦d)),(a¦(X¦¦c¦¦d))}

Fig.5: An example of sets of corresponding modules.

A distribution of roles between the participating event goals is legal if the distribution of reduction types, states and identifiers of the corresponding modules is legal for all sets of corresponding modules.

We define the procedure for checking the distribution for a particular set of corresponding modules from the point of one of the modules of the set. The distribution from the aspect of each module of the set must comply with the following rules:

1. If the reduction type of a module is "no-reduction", any distribution is legal from the point of the module.

2. If the reduction type of a module is "on-the-spot", the corresponding modules must not have this property, unless they have the same reduction identifier.

3. If the reduction state of a module is "pending", the required reduction must be trivial.

4. If the reduction type of a module is "observed", there must exist a corresponding module with the reduction type "on-the-spot".

5. If the reduction type of a module is "strong-common" and its reduction state is "pending", the module must be a whole participating event goal.

If the distribution of roles is legal, the event goals are unified by a procedure, which differs from the default procedure for unifying atomic goals with clause heads only in the way of unifying variables, which remain unassigned in the event: The corresponding variables are not unified into a single variable, but retain their identities to prevent tight coupling of loosely coupled modules.

The modules of the event goals, which have a corresponding module with the reduction type "on-the-spot" and are able to propagate results of an "on-the-spot" reduction, are assigned this property.

4. The Problem of Verification

The ease of verification of specifications in the new language depends on the nature of procedures for execution of goals. For the case of the proposed default procedures we provide some basic guidelines, how clauses of the language could be converted into the familiar and widely treated form with just atomic event and non-event goals, the classical parallel and sequential composition, guarded sequences of goals and synchronous events, which require just pure unification.

The behaviour of a node can be represented as a set of possible execution sequences of "on-the-spot" reduction goals and event goals. If a reduction goal is executed simultaneously with an event goal, it is not represented separately. As goals are not executed more than once, the set is finite and so are the sequences. Hence, they can be specified with the classical composition operators and guards. Each event goal is then replaced by an OR composition of all its forms, that can be generated by grouping and permutation of submodules. Changing any term into an atomic goal is just a syntactic operation, but an exotic component of the language still remains, namely the procedure for execution of event goals.

The problematic part of the proposed default execution procedure is the role distribution checking part. It involves checking and setting of the three implicit variables, associated with each submodule of the participating event goals: the reduction type, state and identifier. When changing an event goal into an atomic goal, its modular structure must be retained, so that every submodule can explicitly be attached the three reduction variables. As the values of the variables are passed between goals, every variable must have its input and its output copy. Analogously, each atomic non-event goal must be attached a variable for generation of its reduction identifier.

5. A Simple Form of the Language and Some Examples

An elegant and detailed syntax for the language is beyond the scope of the paper. To be able to provide some examples, just a very simple form of the language is proposed informally.

A specification is a set of Horn clauses with the following conventions:

a) All composition operators (¦, ¦¦, /, //) should be used as infix operators. If all types of composition operators are treated as one, a clause's body is a hierarchical sequence of modules. For each composition operator it is obvious, which is the sequence, it helps to create. Each module of a sequence is explicitly or implicitly followed by its belonging composition operator. The presence of a composition operator may be implicit, if it belongs to the last module of a sequence and no comment needs to be attached to it. If there is a comment, attached to a module, it may (from the aspect of the syntax) also be treated as attached to the belonging composition operator.

In a comment, attached to a composition operator, various subsequences of the body can be defined, relatively to the position of the operator, with the following syntax:

B: the body.

M: the module, belonging to the composition operator.

S: the sequence, created by the composition operator.

S(n); n:(0),1,2..: S(0) = S. S(n+1) is the sequence, to which S(n) belongs as the rightmost element of its left subsequence.

LS(n): the left subsequence of the sequence S(n). The left subsequence of S is the subsequence to the left of the composition operator.

RS(n): the right subsequence of the sequence S(n). The right subsequence of S is the subsequence to the right of the composition operator.

(nm)X: the subsequence of a sequence X, starting with the n-th element and ending with the m-th element of the sequence X. The constant s denotes the size of a sequence X. Note: If X is a sequence with a single element, (11)X denotes the first element of the element, not the element, and (mn)X denotes a subsequence of the element. Sequences with a single element, which is again a sequence with a single element, are forbidden.

b) Sets of modules, referred to in comments, may be constructed by intersection (.), union (+) and difference (\) from the following simple sets:

X: the set of all modules, belonging entirely to a sequence X.

WX: the module, which covers exactly the whole sequence X.

UX: the set of all modules, containing at least a part of a sequence X.

E: the set of all explicit modules of the body.

I: the set of all implicit modules of the body.

A: the set of all atomic modules of the body.

C: the set of all compound modules of the body.

G: the set of all modules of the body, which are goals.

RG: the set of all modules of the body, which are reduction goals.

EG: the set of all modules of the body, which are event goals.

c) Each explicit module may be followed by a comment #...#, declaring the non-default properties of sets of its submodules and itself. If a property of a particular module is defined in a comment, attached to an explicit module, and again in a comment, attached to one of its explicit submodules, the first declaration is ignored. All specified sets of modules are implicitly in intersection with M.

The non-predefined properties of modules are: to be a reduction goal, to be an event goal, the reduction type and, if the module is an event goal, the required number of participants of the event, in which it will be executed.

d) Each composition operator may be followed by a comment {..}, enumerating some execution-ordering relations in the form S1<S2 (the goals of the set S1 must be executed before the goals of the set S2). All specified sets of modules are implicitly in intersection with G.

e) Each module in an exclusive composition may be followed by a comment [..], specifying its guard. All specified sets of modules are implicitly in intersection with G.

f) The defaults are those, proposed earlier in the paper, plus:

- Modules are not event goals.
- Atomic modules are reduction goals with reduction type "on-the-spot".
- Compound modules are not reduction goals.
- Execution-ordering relation of ¦ and / operators: {A.RG.LS<URS}.
- Execution-ordering relation of ¦¦ and // operators: {}.
- Guards: [A.RG.(11)M], if the module is compound, or [RG.M], if the module is atomic.

```
system:- sender¦¦receiver1¦¦receiver2.

sender:- ((wait1¦¦wait2)#A: (event goal = true;
                              reduction state =
                                no-reduction;
                              participants = 2)#
         ¦{EG.LS<EG.RS}
          (a(X)¦¦b(Y)¦¦c(Z)
          )#M: event goal = true#
         ).
a(X):-..........
b(X):-..........
c(X):-..........

receiver1:- ((c(X)¦¦a(Y)
             )#WM: event goal = true;
               A: reduction type = observed#
            ¦()
             wait1#M: (event-goal = true;
                       reduction state =
                         no-reduction;
                       participants = 2)#
            ¦{LS<RS}
             process1(X,Y)
            ).
process1(X,Y):-.........

receiver2:- ((a(X)¦¦b(Y)
             )#WM: (reduction type =
                       strong-common;
                     reduction state = pending);
               A: reduction type = observed#
            ¦()
             wait2#M: (event goal = true;
                       reduction state =
                         no-reduction;
                       participants = 2)#
            ¦{LS<RS}
             process2(X,Y,Z)
            ).
process2(X,Y,Z):-.........
```

Fig.6: Distribution of subsets.

With these defaults, protocols can be specified entirely in the classical style and (with a slightly modified execution procedure for events) the language used as a dialect for the event-ordering part of LOTOS. Next, we provide some motivation examples for the new features of the language.

Example in Fig.6: A set of data-items is sent to a community of modules, so that every module receives exactly the items of its personal interest in a single event. The sender produces data and waits for creation of the receivers. Whenever all the data-items, interesting for a particular receiver, are generated, they are transmitted and, because of the fairness requirements, the receiver actually receives them.

```
system:- active1¦¦active2¦¦passive.

a(X):-..........
b(X):-..........

active1:- (( a(X)#M: reduction type = observed#
            ¦¦a(Y)
            ¦¦b(Z)
           )#WM: event goal = true#
          ¦(LS<RS)
           process1
          ).
process1:-..........

active2:- (( a(X)
            ¦¦a(Y)#M: reduction type = observed#
            ¦¦b(Z)#M: reduction type = observed#
           )#WM: event goal = true#
          ¦(LS<RS)
           process2
          ).
process2:-..........

passive:- (( a(X)
            ¦¦a(Y)
            ¦¦b(Z)
           )#WM: event goal = true;
             A: reduction type = observed#
          ¦(LS<RS)
           process3
          ).
process3:-..........
```

Fig.7: Distribution of data from several sources and the concept of roles.

Example in Fig.7: A synchronous event is specified, involving collection of data-items from several sources, merging of data into a compound message and its dissemination to all modules of the system. **active1** and **active2** will wait for each other to execute the first event, but will not wait for the passive observer, if its event goal is created to late. If the event was asynchronous, allowing an unlimited number of participants, the passive observer could receive the message regardless of the execution speeds. There are three roles in the event (a, a and b), the first played by **active2** and the others by **active1**.

```
sender:- (Address#M: event goal = true#
         ¦(LS<URS)
          message
         )#WM: event goal = true;
           A: reduction state = no-reduction#.
```

Fig.8: Receving an address and sending a message to the address.

The task in example from Fig.8 is to receive the address, on which a particular message is to be sent, and to send the message.

6. Conclusions

The basic object paradigm of Prolog-type languages is creation and deletion of modules. Some languages of the family (e.g. DP) introduce parallel/sequential composition and explicit communication, what makes them suitable for specification of communication protocols.

The main contribution of the paper is separation of the "module" concept from the concept of "goal", so that a module may participate in several goals (compound modules). Some syntactic enhancements have been proposed for the language of Horn clauses with parallel/sequential composition of atomic event and non-event goals, which in the light of the new operational semantics provide for more concise specification of protocols, particularly those requiring multiple operations on a complicated piece of data. The core idea of the new language is full exploitation of the structure of compound modules, which is usually much more simple than the syntax of atomic modules.

Three independent roles of composition operators have been identified: In the module-grouping and module-ordering role they facilitate specification of sets, subsets, sequences and subsequences of modules. In the execution-ordering role they facilitate specification of the execution order of pending goals. In the selection role they facilitate specification of guarded commands and process disruption.

The concept of reduction has been separated from the concept of event by introducing independent promotion of modules into reduction goals or into event goals. Reduction goals have as well been assigned the execution-ordering role.

To indicate to which extent the execution of a reduction goal depends on execution of an event goal, four reduction types have been introduced: on-the-spot reduction, strong common reduction, weak common reduction and observed reduction. The most original one is the observed reduction, which could serve for specifying observation of exit results, for specifying events with a limited number of roles, for reduction of the computational effort, for reduction of non-determinism and for implementation of monitors of the overall system activity.

The paper indicates, how the architectural and the behavioural aspect of a system can be unified into a single semantic model and efficiently specified by a simple language.

References

[1] R.Milner: A Calculus of Communicating Systems, Springer verlag, LNCS 92, Berlin 1980

[2] C.A.R.Hoare: Communicating Sequential Processes, Communications of the ACM, vol.21, no.8, 1978, pp.666-677

[3] E.Brinksma: A Tutorial on LOTOS, in "Protocol Specification, Testing, and Verification, V", M.Diaz ed., North-Holland 1986, pp. 171-194

[4] L.M.Pereira, F.C.N.Pereira, D.L.D. Warren: User's Guide to DECsystem-10 PROLOG, Occasional Paper 15, Dept.of AI, Edinburg 1979

[5] E.Y.Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT TR-003 (1983)

[6] L.Monteiro: A Proposal for Distributed Programming in Logic, in "Implementations of Prolog", J.Campbell ed., Ellis Horwood 1984

[7] L.M.Pereira, R.Nasr: Delta-Prolog: A Distributed Logic Programming Language, Departamento de Informatica, Universidade Nova de Lisboa, 1985

[8] A.Porto: Two-level Prolog Departamento de Informatica, Universidade Nova de Lisboa, 1985

[9] M-Prolog, Language Reference Manual, Institute for Co-ordination of Computer Techniques, Budapest, March 1983

ISMM

CALL FOR PAPERS

ISMM International Conference

MINI AND MICROCOMPUTERS

From Micros to Supercomputers

December 14-16, 1988
Miami Beach, Florida

SPONSORED BY
The International Society for Mini and Microcomputers (ISMM)
Technical Committee on Computers

SUPPORTED BY
Department of Electrical and Computer Engineering, University of Miami, Coral Gables, Florida, U.S.A.
Modcomp, an AEG Company, Fort Lauderdale, Florida, U.S.A.

LOCATION
Hotel Hilton Fontainbleau, Miami Beach, Florida, U.S.A.

SCOPE
Covers all aspects of computer architecture, organization, and design, artificial intelligence, software systems, and computer applications.

- Computer architecture
- Parallel computers
- Fault-tolerant systems
- VLSI and chip design
- RISC architecture
- Expert systems
- Computer applications
- Parallel and concurrent programming
- Operating systems and databases
- Software engineering
- Real-time systems
- Artificial intelligence
- Supercomputing
- Computer vision
- Robotics
- AI programming and environment
- Software systems
- Local area networks
- Optimizing compilers

Three tutorials will be presented by leading experts in the field. Tentative topics are: Advanced Computer Architecture, Parallel Computers and Artificial Intelligence.

SUBMISSION OF PAPERS

Five copies of a 400-word summary	June 15, 1988
Notification of authors	July 15, 1988
Full papers in camera-ready form	October 1, 1988
Conference	December 14 - 16, 1988

All papers shall be reviewed for possible publication in one of the ISMM journals: International Journal of Mini and Microcomputers, and Microcomputer Applications.

INTERNATIONAL PROGRAM COMMITTEE

G. Rabbat, General Chairman	U.S.A.	M. Kabuka	U.S.A.	V. Oklobdzija	U.S.A
B. Furht, Program Chairman	U.S.A.	P. Liu	U.S.A.	D. Petkovic	U.S.A.
P. Alpar	U.S.A.	E. Luque	Spain	A.M. Salem	Egypt
M. Carapic	U.S.A.	N. Marovac	U.S.A.	B. Soucek	U.S.A.
R. Bisiani	U.S.A.	G. Mastronardi	Italy	W.V. Subbararo	U.S.A.
E. Fernandez	U.S.A.	J.D. Meng	U.S.A.	D. Tabak	U.S.A.
D. Gulch	U.S.A	V. Milutinovic	U.S.A.	M. Tapia	U.S.A.
M. Hamza	Canada	D. Moldovan	U.S.A.	J. Urban	U.S.A.
C.C. Hsu	R.O.C.	S.C. Moon	Korea	F. Vajda	Hungary
T. Ishiko	Japan	B.N. Naumov	U.S.S.R.	P. Visuri	Finland
				M. Vuskovic	U.S.A.

ADDRESS
For correspondence, submission of extended summaries, and to be placed on the mailing list, write to: Dr. B. Furht, Director of Advanced Technology, Modcomp, 1650 West McNab Road, P.O. Box 6099, Mail Stop 850, Ft. Lauderdale, FL 33340-6099, U.S.A.

Please complete and return this form to: ISMM Secretariat, P.O. Box 25, Calgary, Alberta, Canada T3A 2G1

Please send me information concerning:
☐ The International Journal of Mini and Microcomputers
☐ Microcomputer Applications Journal
☐ The International Journal of Robotics and Automation
☐ Expert Systems, Los Angeles, December 1988
☐ Computer Applications in Design Simulation and Analysis, Reno, Nevada, U.S.A., February 1988
☐ Reliability and Quality Control, Los Angeles, December 1988.
☐ Send me call for papers to conferences in the following areas: ______________________________