

# BDL - A Nondeterministic Data Flow Programming Language with Backtracking

A. Schürr

Lehrstuhl für Informatik III, RWTH Aachen  
Ahornstr. 55, D-52074 Aachen, Germany

<http://www-i3.informatik.rwth-aachen.de/people/andy/>

**Abstract:** *Both visual data flow and logic based programming languages have their merits as declarative languages for certain application domains. Combining their concepts to program with data flows and backtracking seems to be a promising idea, which lead to the development of BDL. BDL is a visual data flow programming language with constructs for nondeterministic programming and constraint checking. Its nondeterminism is resolved by using depth-first search and backtracking à la Prolog.*

**Keywords & Phrases:** *data flow programming language, backtracking, nondeterminism, graph rewriting systems.*

## 1. Introduction

Data flow languages (DFL) are one of the most popular branches of visual programming languages. They are used for purposes like image processing [5], data mining with forms [22, 18], and construction of control panels for laboratory instruments [14]. A non-exhaustive list of their *main advantages* looks like follows:

1. DFLs enforce a programming style, where all flow of data between subprocesses and all possibilities for parallel execution of subprocesses are well-documented.
2. Some DFLs — like LabView [24] — give excellent support for rapid prototyping and user interface construction activities.
3. Constructed visual data flow programs are often readable for end users, thereby simplifying communication between software developers and software users.

These statements are affirmed by experience reports “from the real world”. Baroth and Hartsough mention for instance in their case study [4] an average reduction of software development time by at least a factor 4, compared with the usage of nonvisual application domain specific programming languages and tools.

Nowadays existing DFLs differ with respect to many properties. The only survey paper for DFLs [10], we are aware of, compares 15 different DFLs with respect to characteristics like style of representation, available means of abstraction, offered support for iteration, higher-order functions, type concepts, execution modes, and so forth. It is our opinion that additional classification and comparison efforts are necessary to clarify the value of certain characteristics for a given application domain and to identify and implement new interesting combinations of properties, which are not yet part of any available DFL.

As a consequence, we would like to be able to prototype new versions of DFLs in a rather short period of time. At this point we have to mention the work of Fukanaga, Kimura, and Pree published in [8]. They developed a framework for the implementation of new DFLs, which are variants of the well-known DFL Show-and-Tell [13]. This framework is realized in C++ and was already used for the development of a new variant of Hyperflow [12], called ProtoHyperflow. It facilitates the “low-level” task of implementing DFLs in C++, but not their design on a “higher level”.

This was the starting point for our work, which resulted finally in the design and prototypical implementation of a data flow programming language with built-in backtracking mechanisms. The overall goal of this work may be summarized as follows: Construction of a *framework* that supports the *specification* of syntax and semantics of visual (data flow) programming languages and which generates corresponding editors and interpreters from these specifications without any needs for low-level programming activities [20].

Our work is based on the existence of the *graph grammar programming environment* PROGRES [21]. It supports editing, analysis, interpretation, and compilation of graph grammars (or more precisely programmed graph rewriting systems). It has already been used for the development of visual software engineering tools [19] as well as for the specification of the visual database query language GOQL [1]. The most prominent features of PROGRES are (1) a type concept based on EER-like graph schemata for directed attributed graphs, (2) the declarative definition and incremental evaluation of derived attributes and relationships, as well as (3) its depth-first search and backtracking mechanisms that deal with the inherently existing nondeterminism of any rule-based language.

Rather recently, we developed PROGRES specifications for a number of related data flow programming languages, including a variant of Show-and-Tell (without bounded sequential iteration and layout-dependent parameter binding). PROGRES graph schemata were used to define the overall structure of data flow programs, derived attributes were used to model flows of data, and graph rewrite rules were the appropriate means for defining syntax-preserving editing operations as well as step-wise execution of regarded data flow programs.

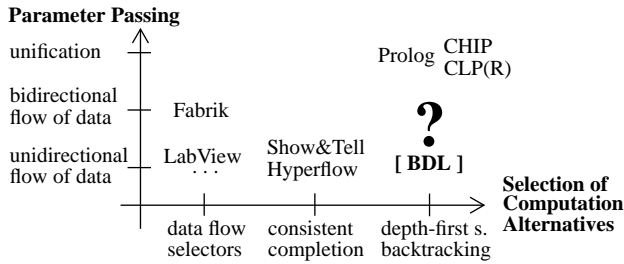


Fig. 1: Logic and data flow programming languages

After some time we recognized that we didn't make any use of PROGRES's backtracking mechanisms. This is a consequence of the fact that not a single DFL we are aware of allows for the definition of nondeterministic processes and resolves nondeterminism by means of depth-first search and backtracking à la Prolog. Figure 1 is an attempt to make the *lack of a whole subclass of DFLs* more apparent. It classifies programming languages with respect to the following two properties:

1. *Parameter passing model*: varies between a unidirectional data flow in traditional DFLs and a bidirectional unification-based data flow in logic-based languages.
2. *Selection of computation alternatives*: well-known solutions are (a) the use of selectors and filters in traditional DFLs which deactivate certain data flows, (b) the consistent completion model of Show-and-Tell, where flows of data are deactivated which would produce inconsistent target box values, (c) the deterministic depth-first-search and backtracking model of Prolog, and (d) a depth-first-search and backtracking model with nondeterministic selection of alternatives.

The discussed observation lead us to the following question, the main topic of this paper:

*Is it possible to develop a nondeterministic data flow programming language with built-in depth-first search and backtracking mechanisms and would such a language be useful in practice?*

The answer should be yes, considering the fact that visual DFLs are closely related to functional programming languages, and the fact that the combination of the functional with the logic programming paradigm is a very active research area for some years (see e.g. [2, 16]).

The rest of this paper studies the raised question in detail. It continues with a very short introduction to the world of data flow programming languages in section 2 and summarizes the "traditional" elements of the proposed **Backtracking Data flow programming Language BDL**.

The following section 3 explains then how control flow nondeterminism may be incorporated into BDL. The same section presents a very compact BDL program, which solves a typical operations research problem of scheduling tasks and resources. It relies heavily on the combination of data flow programming with depth-first-search and backtracking mechanisms.

The following two sections 4 and 5 deal with data flow nondeterminism and cyclic data flows in BDL, respectively. Section 6 and 7 conclude the paper with some remarks concerning the prototypical implementation of BDL and a summary of its main message.

## 2. Basic Elements of BDL

This section recapitulates the basic elements of any data flow programming language and explains their specific representations in BDL. A BDL program is a directed graph with *function boxes* as nodes and *data flow arcs* as directed edges. Data flows along edges from source to target boxes only. Any function box takes the values on its incoming edges as input (without consuming them) and "feeds" its outgoing edges with computed output values. The currently existing version of BDL knows only one type of data, *set of integer*; the Boolean values *true* and *false* are modeled as integer values 1 and 0.

More precisely, BDL programs may contain the following elements (cf. fig. 2):

- User defined functions are subdiagrams (subgraphs), where the flow of data starts at a number of formal input parameters and terminates at a number of formal output parameters. Each parameter has a unique name within the scope of its surrounding user function declaration.
- Function calls are boxes with an appropriate icon and/or name as inscription. All incoming data flows are connected to so-called inlets of a function box, all outgoing data flows are connected to its outlets. The number and names of inlets (outlets) must be identical with the number and names of formal input (output) parameters.
- Data flows connected to the same out parameter (outlet) receive the same values, input data flows connected to the same in parameter (inlet) propagate the union of their value sets.
- Built-in functions, like  $*$  for multiply input values or  $-1$  for decrement input value, take sets of integers as input and produce sets of integers as output. The result of  $\{2, 3\} * \{2, 3\}$  is e.g. the set  $\{4, 6, 9\}$ .

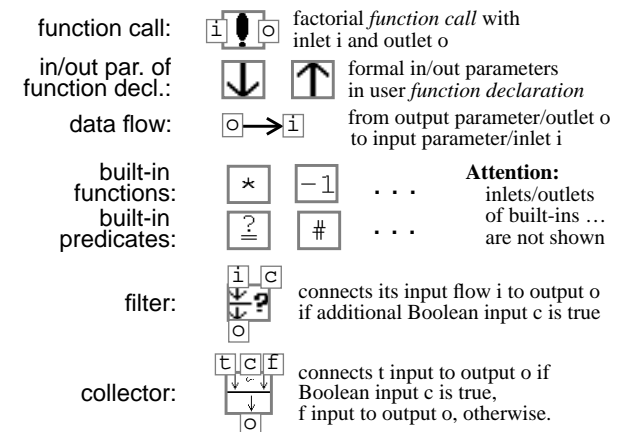


Fig. 2: Used icons and data flow connections

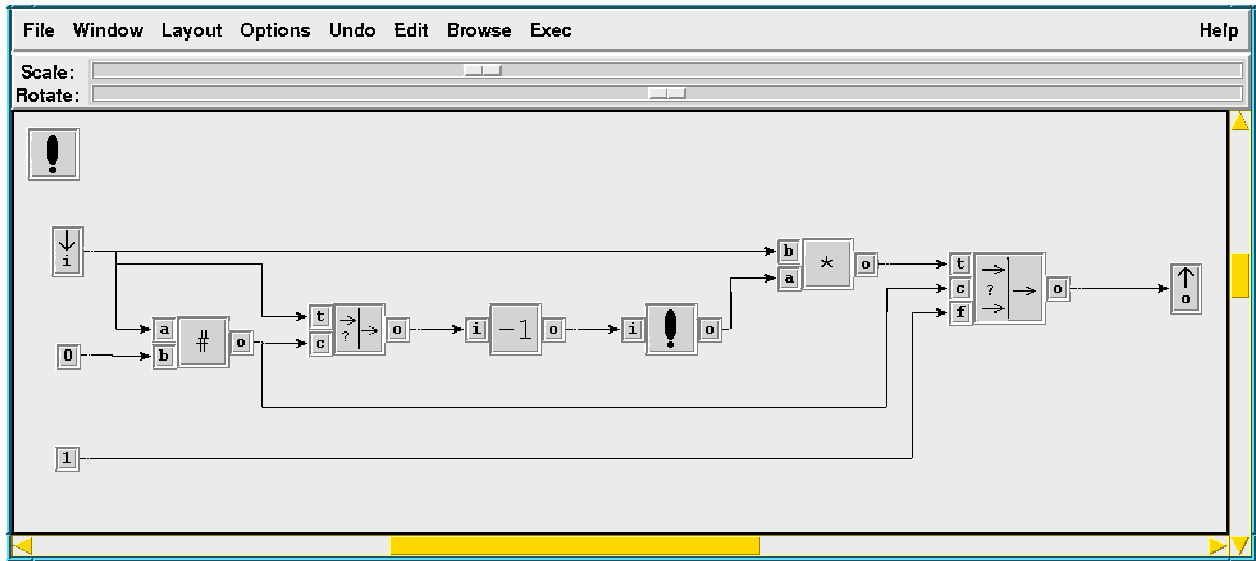


Fig. 3: Screen dump of BDL environment prototype with a definition of the factorial function

- Built-in predicates take sets of values as input and produce a single boolean value as output. The semantics of expressions like  $\{2, 3\} \rho \{2, 3\}$  with  $\rho \in \{=, \#, \dots\}$  is a matter of debate. Our preferred definition looks like follows:  $\text{set } \rho \text{ set}' : \Leftrightarrow \forall e \in \text{set}, e' \in \text{set}' : e \rho e'$ .
- Special filter and collector icons control the flow of data. A filter propagates its input  $i$  to the output, if its input parameter  $c$  has the value `true`; it returns the empty set otherwise. A collector propagates the value of its input parameter  $t$  or  $f$  to the output depending on the fact whether its third input  $c$  is `true` or `false`.

Almost all explained BDL language elements are used in fig. 3 to program once again the factorial function. The displayed screen dump of the prototypical BDL programming environment implementation shows the usual recursive definition. Its execution proceeds as follows for a given input value  $n$  greater than 0:

1. Create a fresh copy of the displayed diagram and propagate the value  $n$  to its in parameter  $i$ .
2. Compare the diagram's actual input value with 0 and use the  $\#$  comparison result `true` to pass the input value  $i$  through the filter.
3. Decrement the output value of the filter, which is still  $n$ , and expand the recursive function call by creating another copy of the subdiagram with the decremented input value  $n-1$ .
4. Complete the recursive call of the factorial function and multiply its result with the old input value  $n$ .
5. The computed result is finally propagated through the  $t(\text{true})$  entry of the diagram's collector to the out parameter  $o$ .

We have to add for reasons of completeness that recursion terminates as soon as the subdiagram's input is 0, i.e. the

predicate  $\#$  delivers the result `false`. This causes the filter to propagate no value to its output and prohibits thereby the evaluation of the decrement function and the recursive call of  $!$ . Furthermore, the collector propagates now the other input 1 to its output, which is the proper result of  $0!$ .

Abstracting from the description above, the *execution of a BDL program* may be modeled as a process which extends and re-evaluates a data flow graph step by step. It starts with the data flow graph of a selected main program and executes the following two steps as long as the constructed data flow graph changes:

1. (Re-)Evaluate any built-in system function with fresh or changed input data (a nonempty set of data).
2. Expand any call of a user defined function with available input data (nonempty sets) by copying its definition, a subdiagram, and connecting its input and output parameters to the corresponding input/output data flows of the function call.

Please note that the current version of BDL expands user function calls only, if all their input parameters have nonempty sets as values. Future versions might distinguish between mandatory and optional parameters. They should then expand a function call as soon as all its mandatory input parameters have nonempty sets as values.

The PROGRES specification of the sketched evaluation process uses an *incremental attribute evaluation algorithm* to perform step 1 above. This guarantees that built-in functions are evaluated in the correct order (imposed by the flow of data) and that any changes of a function's input data trigger only the re-evaluation of its own body and of all its successor functions. Any remaining degrees of freedom of steps 1 and 2 above in the evaluation order of built-in functions and the expansion order of user defined functions are resolved by using a random number generator.

### 3. Nondeterminism and Backtracking in BDL

Having explained the basic elements of the visual data flow programming language BDL it is now time to proceed to the main topic of the paper, nondeterminism and backtracking. But first of all, we will present the promised running example for the rest of the paper. *Project management problems* like task and resource scheduling or warehouse location optimization are a mixture of data flow computation and constraint solving problems. They are nowadays solved by means of dedicated operations research algorithms written in procedural programming languages, or by programs written in logic-based constraint programming languages like CHIP [23] or CLP(R) [11].

A typical though still simple example of this kind is the following *task and resource scheduling* problem: Given is a set of tasks with precedence relationships between them (this task may not be started before the end of that task). Each task needs certain resources, which are assigned manually. Simultaneously active tasks may - of course - not use the same resource.

The determination of earliest/latest start and end times for all tasks as well as a so-called “critical path” in the network of tasks is a typical data flow computation problem. But what about the treatment of a resource conflict between a pair of tasks  $T_1$  and  $T_2$ ? One program written in the logic-based constraint programming language CHIP solves the problem roughly as follows under the assumption that additional resources are not available (cf. [23] for further details): The program computes first preliminary start and end times for all tasks, selects then a pair of tasks  $(T_1, T_2)$  connected to the same resource, inserts either a precedence relationship from  $T_1$  to  $T_2$  or from  $T_2$  to  $T_1$ , recomputes changed start and end times, and repeats this process as long as necessary. Time limits for the whole project or for single tasks may have the effect that the computed earliest possible start time of  $T_1$  or  $T_2$  is greater than its own latest possible end time. This kind of inconsistency causes backtracking, and the program starts to reverse the directions of inserted precedence relationships.

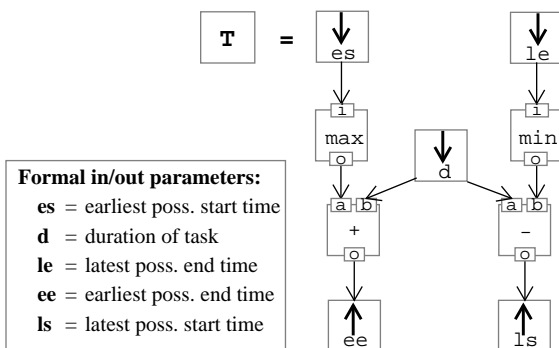


Fig. 4: Definition of function  $T(ask)$

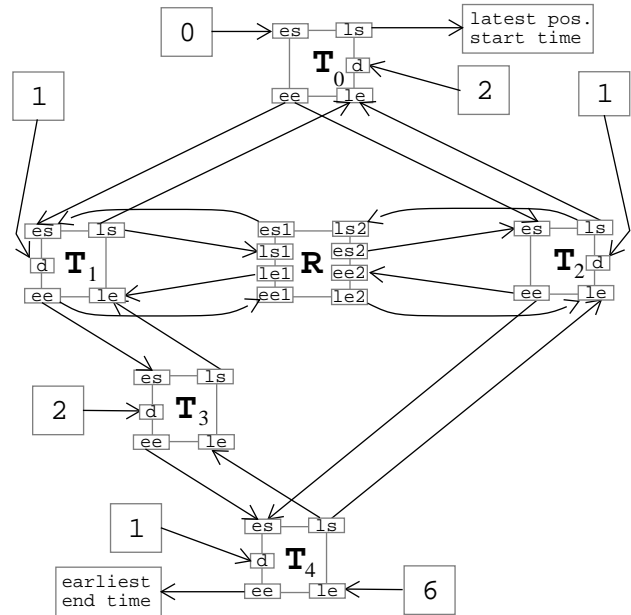


Fig. 5: Main program for a scheduling problem

We will now present a BDL program which implements exactly the algorithm described above. It consists of two user function definitions - one for tasks and one for resources - and a main program, which encodes the dependencies between tasks and resources as appropriate flows of data. The user function  $T(ask)$  receives three input parameters and computes two output parameters. Its input parameters are a set of earliest possible start times  $es$ , estimated duration  $d$ , and a set of latest possible end times  $le$ . Its outputs are the earliest possible end time  $ee$  and the latest possible start time  $ls$ . They are computed as follows:

$$ee := \max(es) + d, \quad ls := \min(le) - d.$$

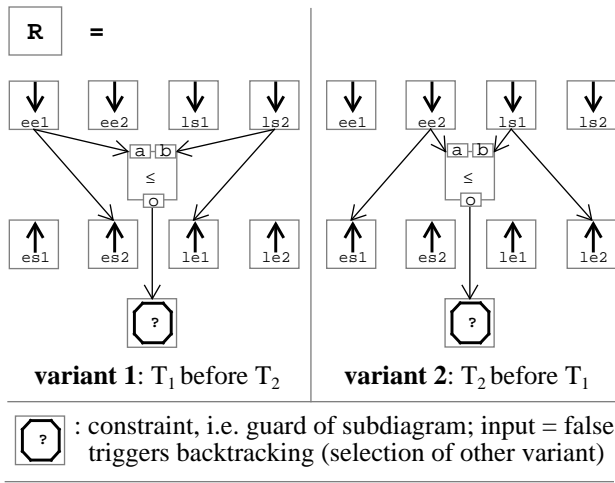
Any precedence relationship from a task (function call)  $T_1$  to a task (function call)  $T_2$  is then translated into a pair of data flow edges, which connect  $T_1.ee$  to  $T_2.es$  and  $T_2.ls$  to  $T_1.le$ , respectively. Fig. 5 contains the corresponding main program for a project with five tasks  $T_0$  through  $T_4$  and the following precedence relationships:

$$T_0 < T_1 < T_3 < T_4 \quad \text{and} \quad T_0 < T_2 < T_4.$$

Please note that the sample data flow program contains two input parameters which are actually connected to two input data flows and receive, therefore, a set of values instead of a single value. These are

$$T_4.es := \{ T_3.ee, T_2.ee \}, \\ T_0.le := \{ T_1.ls, T_2.ls \}.$$

All other input parameters are either connected to a single output parameter or a single constant value. The  $d$  parameters of all  $T(ask)$  function calls receive for instance either the constant value 1 or 2 (time units). And the earliest possible start time  $es$  of the first task  $T_0$  is set to 0, whereas the latest possible end time  $le$  of the last task  $T_4$  is set to 6.



**Fig. 6: Definition of function  $R(\text{resource})$**

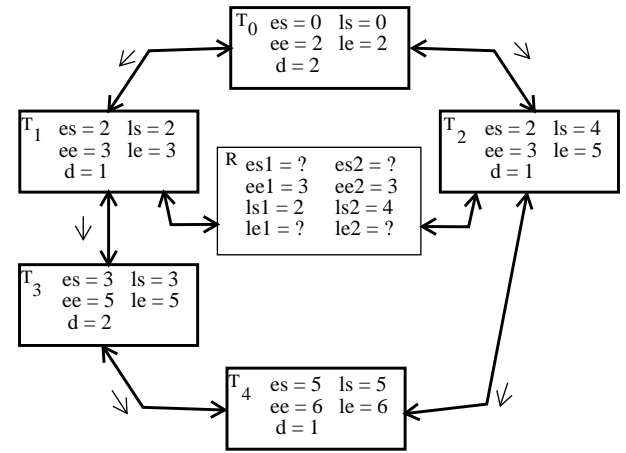
The not yet explained call of function  $R(\text{resource})$  in the centre of fig. 5 models the fact that both  $T_1$  and  $T_2$  need the same resource  $R$ . Its definition is displayed in fig. 6. It is no longer a single subdiagram, but a set of (two) subdiagrams, which have the same input and output parameters, but different flows of data between them.

The  $R(\text{resource})$  function with its two cases has a corresponding input parameter for any output parameter of its two related  $T(\text{ask})$  functions, and it has a corresponding output parameter for any input parameter of the two involved  $T(\text{ask})$  functions. The overall idea of its implementation may be described as follows:

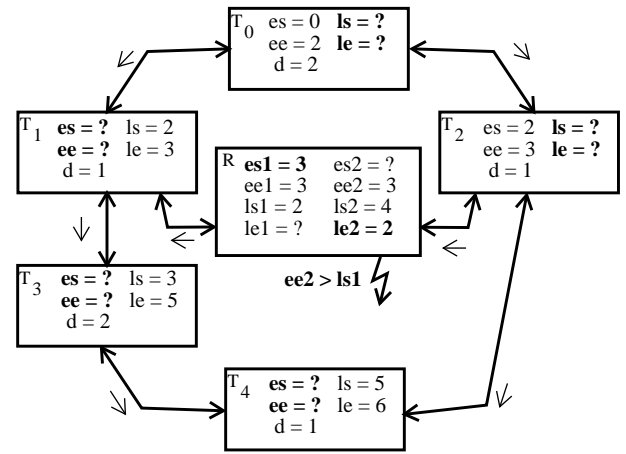
1. The left variant may be selected if and only if the earliest possible end time of task  $T_1$  ( $R.\text{es}1 = T_1.\text{es}$ ) is less equal than the latest possible start time of task  $T_2$  ( $R.\text{es}2 = T_2.\text{es}$ ), i.e. if the depicted timing constraint is fulfilled. It connects task  $T_1$  in the same way to task  $T_2$  as a precedes relationship would connect  $T_1$  to  $T_2$  ( $T_2.\text{es} := T_1.\text{ee}$ ,  $T_1.\text{le} := T_2.\text{ls}$ ).
1. The right variant simply exchanges the roles of tasks  $T_1$  and  $T_2$ , respectively.

The following fig. 7 displays three important intermediate states of our main program's execution trace. Fig. 7.a displays the data flow graph after the expansion (evaluation) of all  $T(\text{ask})$  functions. Functions with expanded bodies are collapsed into single nodes with bold borders and all data flow edges between two function nodes are collapsed into a single (bidirectional) edge, in order to keep the diagram legible. Small arrows have been added as decorations of data flows. They illustrate the established precedence relationships between tasks. The text labels of function boxes display their input and output parameters together with their current values (question marks are used for still uncomputed values). The displayed parameter values reveal that the (estimated) time for the execution of all tasks is 6 units, as long as the possibly existing resource conflict is not taken into account.

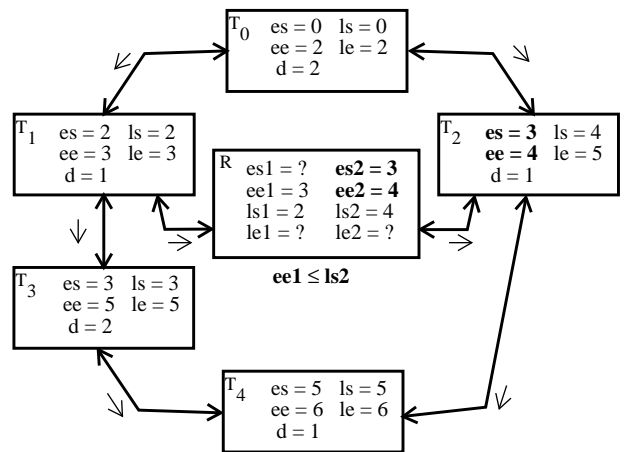
**a) Task network before resource conflict check:**



**b) Remove resource conflict with right task  $T_2$  precedes left task  $T_1$  (fails):**



**c) Remove resource conflict with left task  $T_1$  precedes right task  $T_2$  (succeeds):**



**Fig. 7: Evaluation of Task Network**

The next snapshot of fig. 7.b shows the effect of unfolding the  $R(RESOURCE)$  function call by inserting the right subdiagram of fig. 6. The resulting modification of the data flow graph invalidates a number of previously computed function and parameter values. Some of these values are immediately recomputed as a side effect of the check whether the inserted constraint holds (affected parameter values are displayed in bold figures). Others may or may not be recomputed depending on the fact whether we are using a demand driven lazy or an eager data flow re-evaluation algorithm (the BDL's prototype uses a lazy implementation). The constraint check  $ee2 \leq ls1$  fails as indicated and triggers backtracking, i.e. the expansion of the  $R(RESOURCE)$  function call has to be reconsidered.

This results in the execution state of fig. 7.c, where the left subdiagram of fig. 6 was used to expand the regarded  $R(RESOURCE)$  function call. The constraint of this subdiagram holds due to the fact that it is possible to perform task  $T_2$  after task  $T_1$  in parallel with task  $T_3$ , i.e. the latest possible start time of  $T_2$  is greater equal then the earliest possible end time of  $T_1$ .

Adding constraints and backtracking to BDL requires some modifications of its main *interpreter loop*. The new version executes still as long as the constructed data flow graph changes, but performs now the following steps:

1. (Re-)Evaluate (needed) built-in functions with changed or new input data.
2. Check constraints and trigger backtracking if at least one constraint is violated. Backtracking returns to the latest nondeterministic user function expansion (cf. step 3) and chooses a still remaining alternative.
3. Expand user function call(s) with available input data. Inserted subdiagrams are chosen nondeterministically (randomly) in the case of function definitions which consist of more than one subdiagram (as in fig. 6).

Please note that constraint checking and backtracking is not always as simple as in the explained program trace, where the insertion of a subdiagram causes immediately the violation of its own constraint. In the general case it may happen that an inserted constraint exists for a long time, until its value changes to `false`. The detected constraint violation and the triggered backtracking process has then to undo one function expansion after the other, until it returns to a point in the past with still unexplored expansion alternatives. This may or may not be the expansion of the function that inserted the violated constraint.

#### 4. Data Flow Nondeterminism in BDL

Until now we considered only the case, where nondeterminism was caused by function definitions consisting of more than one subdiagram. This is what we called "*control flow nondeterminism*". A new selector function gives programmers now the possibility to model *data flow nondeterminism*, too. It takes a set of values as input and forwards a (randomly) selected element of the input set to its output.

The *interpreter loop* has to be modified as follows to take the new source of nondeterminism into account:

1. (Re-)Evaluate (needed) built-in functions with available input data. The evaluation of a selector function is the selection and propagation of one of its input values to its output port.
2. Check constraints and trigger backtracking if at least one constraint is violated. Backtracking returns to the latest nondeterministic user function expansion (cf. step 3) or input element selection step (cf. step 1).
3. Expand user function call(s) with available input data. Inserted subdiagrams are chosen nondeterministically (randomly) in the case of function definitions which consist of more than one subdiagram.

Using the just introduced selector function we could modify the example of section 3 as depicted in fig. 8. The input parameter  $le$  of task  $T_4$  is now either 5 or 6 or 7.

Taking into account that the actual implementation of the selector function does not pick elements randomly from the input set, but returns always the smallest remaining element, the modified program is executed as follows:

1.  $T_4.le$  is set to 5.
2. All  $T(ask)$  function calls are expanded and their output parameters are computed.
3. The output parameter  $ee$  of  $T_4$  has now the same value 6 as in fig. 7.a.
4. This violates constraint  $T_4.ee \leq 5$  of fig. 8.
5. Backtracking starts and sets  $T_4.le$  now to 6.
6. Afterwards, the execution process may continue as depicted in fig. 7 of section 3.

It is worth-while to mention that backtracking reconsiders once met decisions in their *chronological reverse order*. This has the consequence that the sketched violation of the first  $R(RESOURCE)$  function call expansion in fig. 7.b is not resolved by simply setting  $T_4.ee$  to 7, but by exploring first the remaining expansion alternative for  $R$ . Furthermore, we have to admit that a future version of BDL should allow for the definition of *application specific selection strategies*. Choosing to be inserted subdiagrams always randomly and to be forwarded integer values always in increasing order is a rather ad hoc approach.

Ellipsis "... " is placeholder for upper part of Fig. 4.

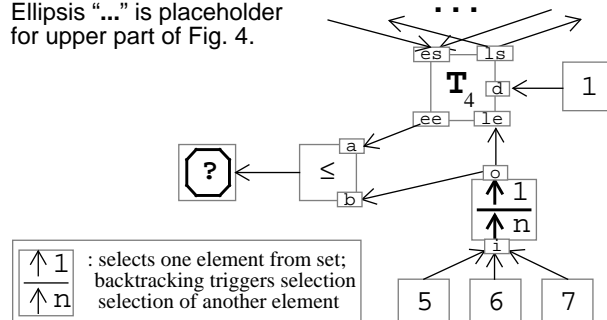


Fig. 8: Modified Fig. 5 with additional constraint and data flow nondeterminism

## 5. Cyclic Data Flows and Iteration in BDL

Until now, we have always made the implicit assumption that constructed data flow graphs are acyclic and that any kind of needed iteration process is defined by using recursive functions. It is the topic of this section to demonstrate that the construction and (fix-point) evaluation of cyclic data flows would be useful in practice and to show the relationships between cyclic data flows and iteration.

First of all we have to distinguish two kinds of iteration, which are usually called parallel iteration and sequential iteration, respectively. *Parallel iteration* means the simultaneous execution of a function call for all element of a given list or set of input elements. All computed results are returned as another set or list of elements. This kind of iteration is for instance available in Show-and-Tell [13] or Prograph [7], and it was already introduced in section 2 for built-in functions like \* or +.

It is clear that the concept of parallel iteration can easily be added to any DFL without destroying the concept of stateless programming. But this is no longer true for *sequential iteration*. A programmer's intention behind this kind of iteration is to evaluate the same "piece of code" (subdiagram) again and again with its own old output values as new input values, until a certain condition becomes true. It is a matter of debate whether a DFL should really support this kind of iteration, but it is a matter of fact that (almost) all really used DFLs support sequential iteration.

There are at least two main approaches how sequential iteration is added to a DFL. The first one is based on creating a *fresh copy* of the iterated subdiagram for each execution cycle. It avoids the construction of cyclic data flows

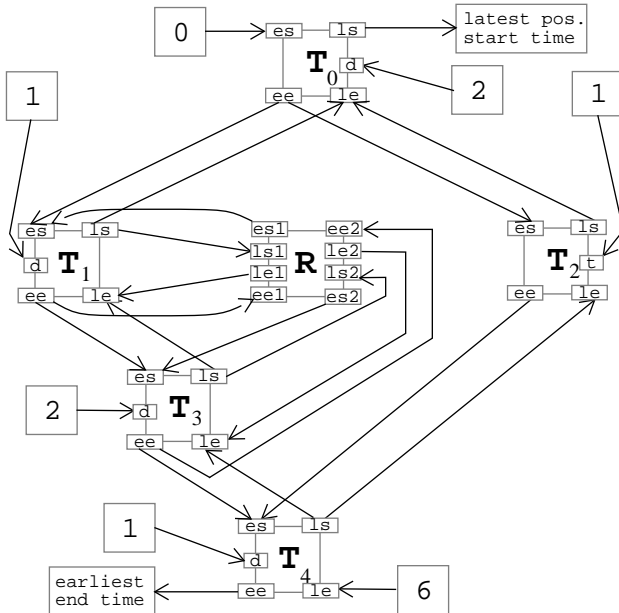


Fig. 9: BDL data flow program with potentially cyclic data flow graph at runtime

and reflects the opinion that iteration is just a special form of recursion (tail recursion). The second one allows for the construction of *cyclic data flows* and deals with them by introducing the concept of evaluation states. In the most general case, any function or variable possesses an associated stream of values instead of a single value. New stream entries for function outputs are then computed based on old stream entries for function inputs (as in Hyperflow [12]). In the simplest case, data flow cycles are broken by introducing special loop variables, which possess a pair of values (old, new) instead of a single value. This is the currently implemented sequential iteration concept of BDL.

The main reason for favoring the approach based on *loop variables* was our believe that cyclic data flow programs are often the natural solution for a given problem. Consider for instance a slightly modified version of the task scheduling problem in section 3, where tasks  $T_1$  and  $T_3$  (and not  $T_1$  and  $T_2$ ) share the same resource. A straightforward translation of the new situation would result in the construction of the data flow program in fig. 9. Its execution may or may not construct a cyclic data flow graph as an intermediate computation result depending on the fact whether the call of  $R$  is first expanded to the right subdiagram or to the left subdiagram in fig. 6. Such a cycle may be broken by treating all in parameters of  $R$  functions as loop variables.

The modified version of the interpreter's main loop for loop variables has about the following (final) form:

1. (Re-)Evaluate all built-in functions with respect to old loop variable values (initial values are empty sets). Halt execution in the case of any data flow cycles which are not broken by the introduction of loop variables.
2. Replace all old loop variable values by the (union of) the values on incoming data flows simultaneously.
3. Check constraints and start backtracking if necessary.
4. Expand user function call(s) with available input data.

## 6. Implementation of BDL

It was already mentioned that the BDL programming environment prototype - shown in fig. 3 of section 2 - is not hand-coded, but automatically derived from a high level *graph grammar specification*. The developed specification is not just a description of the language BDL itself, but defines the complete functionality of its accompanying programming environment. This includes aspects like syntax-directed editing of BDL (sub-)diagrams, browsing and zooming through diagrams as well as stepwise execution.

Writing and testing the specifications for both BDL and a subset of Show-and-Tell took about 4 weeks. The resulting BDL specification has a length of 30 pages with 23 pages being reused from the preceding specification of Show-and-Tell. The *generated BDL prototype* consists of 32.000 lines of C code on top of our own graph-oriented database system GRAS and the user interface toolkit Tk/Tcl. It supports all presented language elements in this

paper (control and data flow nondeterminism, iteration, etc.) and was used as a test bed for validating the design of BDL. Its existence was especially helpful during the design of BDL's operational semantics, i.e. the definition of the interaction between propagation of values along data flows, updating loop variables, checking constraints, and backtracking. The complete specification of BDL will be part of the next forthcoming PROGRES environment release, which is accessible via its world wide web page <http://www-i3.informatik.rwth-aachen.de/research/progres>.

For further details concerning the underlying philosophy of our graph grammar based language description and prototype generation process the reader is referred to chapter 4 of [19] and to [20]. Further details concerning the used graph grammar specification language PROGRES and its tools may be found in chapter 3 of [19] or in [21].

## 7. Conclusions

Summarizing the main message of this paper, we can answer the question raised in section 1 as follows (based on our experiences with the prototypical implementation of BDL and the presented task scheduling example):

*It is possible to add depth-first search and backtracking mechanisms to a visual data flow programming language. The resulting language should be useful for solving about the same class of problems as functional logic programming languages.*

Nevertheless, it is quite clear that BDL represents just a first attempt to add control and data flow nondeterminism to a DFL by replacing the deactivation of inconsistent sub-diagrams in Show-and-Tell with the new concept of backtracking out of inconsistent states. It would for instance be interesting to combine the new concepts of BDL with the idea of *bidirectional (undirected) data flows* as offered in the DFL programming environment Fabrik [15] and especially in the constraint-based system ThingLab [17]. Such an extension would cover another important aspect of (functional) logic programming languages [2, 16], the existence of parameters which change their role (mode) from in to out parameter and back as needed.

Finally, we have to point out that the prototypical implementation of BDL relies heavily on the rather general and therefore inefficient depth-first search and chronological backtracking mechanism of the specification language PROGRES. The necessary overhead for keeping track of old execution states could be reduced significantly. Furthermore, certain search space reducing techniques of constraint-based programming languages — like forward checking [23] or intelligent backtracking [6] — should be useful for reducing the vast search space of BDL programs, too.

## References

- [1] Andries M., Engels G.: *A Hybrid Query Language for an Extended Entity-Relationship Model*, in: JVLIC, vol. 7, no. 3, Academic Press (1996), 321-352
- [2] Ait-Kaci H., Lincoln P. Nasr R.: *Le Fun: Logic, Equation, and Functions*, in: Proc. 1987 Symp. on Logic Programming, IEEE CS Press (1987)
- [3] Burnett M.M., Goldberg A., Lewis T.G. (eds.): *Visual Object-Oriented Programming - Concepts and Environments*, Manning Publications Co. (1995)
- [4] Baroth E., Hartsough Ch.: *Visual Programming in the Real World*, in: [3], 21-44
- [5] Birchman J.J., Tanimoto S.L.: *An Implementation of the VIVA Language on the NeXT Computer*, in: Proc. VL'90, IEEE CS Press (1990), 177-183.
- [6] Bruynooghe M.: *Solving Combinatorial Search Problems by Intelligent Backtracking*, in: Information Processing Letters, vol. 12, no. 1, 36-39, Elsevier Science Publ. (1981)
- [7] Cox P.T. et al: *Prograph*, in: [3], 45-66
- [8] Fukunaga A.S., Kimura T.D., Pree W.: *Object-Oriented Development of a Data Flow Visual Language System*, in: Proc. VL'93, IEEE CS Press (1993), 134-141
- [9] Glinert E.P. (ed.): *Visual Programming Environments: Paradigms and Systems*, IEEE CS Press (1990)
- [10] Hills D.D.: *Visual Languages and Computing Survey: Data Flow Visual Programming Languages*, in: JVLIC, vol. 3, no. 1, Academic Press (1992), 69-101
- [11] Jaffar J., Michaylov S., Stueckey P.J., Yap R.H.C.: *The CLP(R) Language and System*, in: TOPLAS, vol. 14, no. 3, ACM Press (1992), 339-416
- [12] Kimura T.D.: *Hyperflow: A Visual Programming Language for Pen Computers*, in: Proc. VL'92, IEEE Computer Society Press (1992), 125-132
- [13] Kimura T.D., Choi Y.Y., Mack J.M.: *Show and Tell: A Visual Programming Language*, in: [8], 397-404
- [14] Kodosky J., MacCrisken J., Rymar G.: *Visual Programming Using Structured Data Flow*, in: Proc. VL'91, IEEE CS Press (1991), 34-40
- [15] Ludolph F., Chow Y.Y., Ingalls D., Wallace S., Doyle K.: *The Fabrik Programming Environment*, in: Proc. VL'88, IEEE Computer Society Press (1988), 222-230
- [16] Lock H.C.R.: *The Implementation of Functional Logic Languages*, Oldenbourg Verlag (1993)
- [17] Maloney J.H., Borning A., Freeman-Benson B.: *User-Interface Construction with Constraints*, in: [3], 113-128
- [18] Miyao J. et al.: *Visualized and Modeless Programming Environment for Form Manipulation Language*, in: Proc. VL'89, IEEE CS Press (1989), 99-104
- [19] Nagl M. (ed.): *Building Tightly-Integrated (Software) Development Environments: The IPSEN Approach*, LNCS 1170, Springer Verlag (1996)
- [20] Rekers J., Schürr A.: *A Graph Based Framework for the Implementation of Visual Environments*, in: Proc. VL'96, IEEE CS Press (1996), 148-155
- [21] Schürr A., Winter A., Zündorf A.: *Visual Programming with Graph Rewriting Systems*, in: Proc. VL'95, IEEE CS Press (1995), 326-335
- [22] Tjian B.S. et al: *A Data-Flow Graphical User Interface for Querying a Scientific Database*, in: Proc. VL'93, IEEE CS Press (1993), 49-54
- [23] Van Hentenryck P.: *Constraint Satisfaction in Logic Programming*, MIT Press (1989)
- [24] Vose G.M., Williams G.: *LabView: Laboratory Virtual Instrument Engineering Workbench*, BYTE, vol. 11, no. 9, McGraw-Hill (1986), 84-92