[TW91] A. Trew, G. Wilson: Past, Present, Parallel, Springer, 1991, 126-137

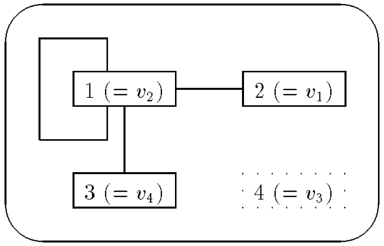
A Syntax of TDL

```
TDLscript ::=
    Defs Subnet
Defs ::=
    ConstDefs FctDefs
ConstDefs ::=
    ε | const ConstDef ;
FctDefs ::=
    function FctId ( FormPars )
    { Net} FctDefs
  | function FctId ( )
    { Net } FctDefs
  | ε
ConstDef ::=
    ConstId = Exp, ConstDef
  | AttrId = Attrs, ConstDef
  | ConstId = Exp
  | AttrId = Attrs
FormPars ::=
    FormPar | FormPar, FormPars
FormPar ::=
    ConstId | NetId | AttrId
Subnet ::=
    subnet SubnetId { Contr Block }
Contr ::=
    ε | control Rst_Anl [ Path ];
Block ::=
    terminal P { Links; Attrs }
    Block
  | Subnet Block
  | ε
Exp ::=
    ( Exp )
  | Exp Binop Exp
  | − Exp
  | Z | # Hex | ConstId
Binop ::=
    + | * | − | /
  | ↑ | mod | <<
Net ::=
    ( Net )
  | for Id = Exp to Exp NextList
    result is Net
  | if BExp then Net else Net
  | let LocalDef in Net
  | Net List
  | [ Exp ] Net
  | substitute ( Net, List, Net )
  | connect ( Net, List, List )
  | join ( Net, Net )
  | processor ( Attrs )
  | node ( Net )
  | FctId ( ParamList )
  | FctId ( )
  | NetId
NextList ::=
    next NetId is Net
  | NextList, next NetId is Net
ParamList ::=
    Exp, ParamList
  | Net, ParamList
  | Attrs, ParamList
  | Exp
  | Net
  | Attrs
LocalDef ::=
    NetId = Net
  | LocalDef, NetId = Net
List ::=
    all | [ Sections ]
Sections ::=
    Sections, Exp
  | Sections, Exp .. Exp
  | Exp
  | Exp .. Exp
```



```
Links ::=
    Links, Links
  | ε
  | P
  | Path
  | ext [ Exp ]
  | Net
Attrs ::=
    Attrs ; Attrs
  | TypeId
  | ptype ProcType
  | memory Exp
  | attrib UserAttr
  | mnode Masternode
  | HELIOS | SYSTEM
  | IO | NATIVE
  | ε
ProcType ::=
    T800 | T414 | T212
  | 68000 | 80286
UserAttr ::=
    AttrId | AttrId [ Exp ]
Masternode ::=
    Rst_Anl [ FileName ]
  | Configure [ FileName ]
BExp ::=
    Exp Relop Exp
  | BExp and BExp
  | BExp or BExp
  | not BExp
  | ( BExp )
  | odd ( Exp ))
Relop ::=
    <= | >= | <
  | > | = | <>
Path ::=
    /SubnetId Path
  | P
NetId ::= Id
FctId ::= Id
ConstId ::= Id
AttrId ::= Id
FileName ::= Id
SubnetId ::= Id
```

A number Z is a sequence of digits. An identifier Id consists of a sequence of letters, digits, '.', and '_', beginning with a letter. A processor name P is either a number or an identifier. Hex is a hexadezimal string consisting of digits and the letters 'a' to 'f' and 'A' to 'F'.



We have chosen this rather complicated way of describing the above graph in order to demonstrate many features of TDL. The same graph is produced by:

connect([4]processor(HELIOS), [1,1,1], [1,2,3]).

5 Conclusion and Extensions

We have presented the syntax and semantics of the topology description language TDL. We have shown that topologies of practical interest can be described in a few lines. Such a description is easy to understand and to modify. Practical experience shows that there is few danger of making errors.

As we already pointed out, TDL has been designed for an environment, where the physical link numbers are not important and may be left implicit. For other environments, an explicit treatment of link numbers may be needed, e.g. for the description of statically connected multiprocessor systems. Since TDL includes the Helios RML (which forces the management of link numbers), TDL may also be used in such environments, but it looses much of its elegance.

Thus, those TDL operations, which add connections, namely connect and join, need to be generalized: in connect(NET, LIST$_1$, LIST$_2$), each element of LIST$_1$ and LIST$_2$ may consist of a processor number and a corresponding link number. Each processor in LIST$_1$ will be connected to the corresponding processor in LIST$_2$ using the specified link numbers.

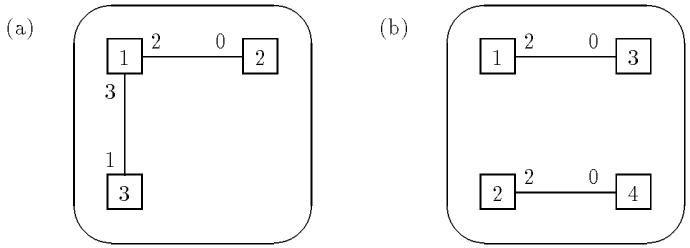


Figure 4: (a) connect([3] processor (), [1::2, 1::3], [2::0, 3::1]),
(b) join([2] processor() :: 2, [2] processor() :: 0)

Similarly, join(NET$_1$ :: LINKNO$_1$, NET$_2$ :: LINKNO$_2$) will connect link LINKNO$_1$ of each processor of NET$_1$ to link LINKNO$_2$ of the corresponding processor of NET$_2$.

Figure 4 shows two examples. The graph depicted in Figure 1 can be generated by:

```
const T800MB = ptype T800; memory #100000; HELIOS;

function matrix(n, m, left, right, up, down, type){
    if n = 1 then vector(m, left, right, type)
    else join(vector(m, left, right, type) :: down,
              matrix(n−1, m, left, right, up, down, type) :: up)}

function vector(n, left, right, type)
    {connect([n]processor(type), [1..n−1 :: right], [2..n :: left])}

subnet /Cluster{
    control Rst_Anl[/Cluster/00];
    terminal 00 {~IO, , matrix(2, 3, 0, 2, 1, 3, T800MB) :: 0, ;
        T800MB; Mnode Rst_Anl[pa_ra.d];}
    terminal IO { ; IO}
}
```

If TDL shall be used for the description of software topologies, the link numbers have to be replaced by names of pipes or software channels.

References

[IN88a] INMOS Ltd.: Transputer Instruction Set: A Compiler Writer's Guide, Prentice Hall, 1988

[IN88b] INMOS Limited: occam 2 Reference Manual, Prentice Hall, 1988

[KL88] F.D. Kübler, F. Lücking: A Cluster Oriented Architecture for the Mapping of Parallel Processor Networks to High Performance Applications, ACM Proceedings of the 1988 International Conference on Supercomputing, St. Malo

[Kr88] O. Krämer-Fuhrmann: Gracia: A Software Environment for Graphical Specification, Automatic Configuration, and Animation of Parallel Programs, ACM Proceedings of the International Conference on Supercomputing, Köln, 1988

[Pa90] Parsec Developments: ParC System – Users's Manual and Library Reference, P.OBox 782, 2300 AT Leiden, The Netherlands, 1990

[Pe89a] Perihelion Software Limited: The HELIOS Operating System, Prentice Hall, 1989

[Pe89b] Perihelion Software Limited: The Network Guide, Perihelios Software Limited, 1989

[St91] Holger Stoltze: Entwurf und Implementierung einer Sprache zur Beschreibung von Netzwerktopologien unter dem Betriebssystem HELIOS, Master Thesis, RWTH Aachen, 1991 (in German)

$\mathcal{L}[\![[\text{EXP}_1..\text{EXP}_2,\ \text{SECTIONS}]]\!]\ G\ \rho :=$
$\quad$ *let* $k_1 := \mathcal{E}[\![\text{EXP}_1]\!]\ \rho, \quad k_2 := \mathcal{E}[\![\text{EXP}_2]\!]\ \rho, \quad \langle V, E, \beta, v_1 : \ldots : v_n, L\rangle\ := G$
$\quad$ *in if* $1 \le k_1 \le k_2 \le n$ *then* $v_{k_1} : v_{k_1+1} : \ldots : v_{k_2} : (\mathcal{L}[\![[\text{SECTIONS}]]\!]\ G\ \rho)$
$\quad\quad$ *else* $\mathcal{L}[\![[\text{SECTIONS}]]\!]\ G\ \rho$

$\mathcal{L}[\![[\text{EXP}]]\!]\ G\ \rho :=$
$\quad$ *let* $k := \mathcal{E}[\![\text{EXP}]\!]\rho, \quad \langle V, E, \beta, v_1 : \ldots : v_n, L\rangle\ := G$
$\quad$ *in if* $1 \le k \le n$ *then* v_k

$\mathcal{L}[\![[\text{EXP}_1..\text{EXP}_2]]\!]\ G\ \rho :=$
$\quad$ *let* $k_1 := \mathcal{E}[\![\text{EXP}_1]\!]\ \rho, \quad k_2 := \mathcal{E}[\![\text{EXP}_2]\!]\ \rho, \quad \langle V, E, \beta, v_1 : \ldots : v_n, L\rangle\ := G$
$\quad$ *in if* $1 \le k_1 \le k_2 \le n$ *then* $v_{k_1} : v_{k_1+1} : \ldots : v_{k_2}$
$\quad\quad$ *else* ε

The expression semantics $\mathcal{E}$ is defined as usual:

$\mathcal{E}$: $\underline{\text{EXP}} \cup \underline{\text{BEXP}} \cup \underline{\text{ATTRS}} \longrightarrow \mathbb{B} \cup \mathbb{Z} \cup Attributes^*$

$\mathcal{E}[\![\text{ID}]\!]\ \rho\ :=$ *if* $\rho(\text{ID}) \in \mathbb{Z} \cup Attributes^*$ *then* $\rho(\text{ID})$

$\mathcal{E}[\![\text{Z}]\!]\ \rho\ := \text{z}$
$\quad$ where Z is a symbol for a constant (in decimal or hexadecimal representation)
$\quad$ and $\text{z} \in \mathbb{Z}$ is the number associated with Z.

$\mathcal{E}[\![\ \text{EXP}_1\ \text{OP}\ \text{EXP}_2\ ]\!]\ \rho := \ (\mathcal{E}[\![\ \text{EXP}_1\ ]\!]\ \rho)\ \text{op}\ (\mathcal{E}[\![\ \text{EXP}_2\ ]\!]\ \rho)$
$\quad$ where $\text{OP} \in \{+, -, *, /, \uparrow, <<, mod, <=, >=, <>, <, >, =, and, or\}$
$\quad$ and op is the semantic function corresponding to the operation symbol OP.

$\mathcal{E}[\![\ \text{OP}\ \text{EXP}\ ]\!]\ \rho :=\ \text{op}\ (\mathcal{E}[\![\ \text{EXP}\ ]\!]\ \rho)$
$\quad$ where $\text{OP} \in \{-, not, odd\}$
$\quad$ and op is the semantic function corresponding to the operation symbol OP.

$\mathcal{E}[\![\ (\text{EXP})\ ]\!]\ \rho :=\ \mathcal{E}[\![\ \text{EXP}\ ]\!]\ \rho$

$\mathcal{E}[\![\ \mathsf{memory}\ \text{EXP};\ \text{ATTRS}\ ]\!]\ \rho :=\ (\text{memory}, \mathcal{E}[\![\ \text{EXP}\ ]\!]\ \rho)\text{: } \mathcal{E}[\![\ \text{ATTRS}\ ]\!]\ \rho$
The other attributes are handled analogously.

$\mathcal{E}[\![\ \varepsilon\ ]\!]\ \rho\ :=\ \varepsilon$

In order to show, how the above definitions work together, we will compute the semantics of an example definition. This will be done "bottom-up". Consider the TDL definition Δ:

$$\begin{array}{l} \mathsf{function}\ \mathtt{test}(n,\ \mathtt{type})\ \{ \\ \quad \mathsf{let}\ P\ =\ \mathsf{processor}(\mathtt{type}) \\ \quad \mathsf{in}\ \mathsf{join}((\ \underbrace{\mathsf{connect}([3]P,\ [1..2],\ [2,2,1])}_{=:E}\ )[n,1],\ P)\} \end{array}$$

For some $y_1 \in \mathbb{Z}$ and $y_2 \in Attributes^*$, let $\rho_1 := \rho_0[\mathtt{test}/\theta,\ n/y_1,\ \mathtt{type}/y_2]$, where ρ_0 is the empty environment, which is undefined for all arguments, and $\theta(y_1, y_2) := \mathcal{N}[\![\Delta]\!]\rho_1$. Moreover $\rho_2 := \rho_1[P/\mathcal{N}[\![\mathsf{processor}(\mathtt{type})]\!]\rho_1]\ =\ \rho_1[P/\langle\{v_1\}, \emptyset, \beta_1, v_1, \varepsilon\rangle]$ where $\beta_1(v_1) = y_2$.
Since

$$\mathcal{N}[\![P]\!]\rho_2 = \rho_2(P) = \langle\{v_1\}, \emptyset, \beta_1, v_1, \varepsilon\rangle, \tag{1}$$

we get

$$G := \mathcal{N}[\![[3]P]\!]\rho_2 = \bigoplus_{i=1}^{3}\langle\{v_1\}, \emptyset, \beta_1, v_1, \varepsilon\rangle = \langle\{v_1, v_2, v_3\}, \emptyset, \beta, v_1 : v_2 : v_3, \varepsilon\rangle, \tag{2}$$

where $\beta(v_i) = y_2$ for i=1,2,3 (Note the renaming of nodes). Furthermore,

$$\mathcal{L}[\![[1..2]]\!]\ G\ \rho_2\ =\ v_1 : v_2 \quad \text{and} \quad \mathcal{L}[\![[2,2,1]]\!]\ G\ \rho_2\ =\ v_2 : v_2 : v_1. \tag{3}$$

With (2) and (3), we get

$G' := \mathcal{N}[\![\ \mathsf{connect}([3]P,\ [1..2],\ [2,2,1])]\!]\ \rho_2\ =\ \langle\{v_1, v_2, v_3\},\ \{\{v_1, v_2\}, \{v_2\}\},\ \beta,\ v_1 : v_2 : v_3,\ \varepsilon\rangle$

and, since $\mathcal{L}[\![[n,1]]\!]\ G'\ \rho_2 = v_{y_1} : v_1$,

$$\begin{array}{l} \mathcal{N}[\![\ (\mathsf{connect}([3]P,\ [1..2],\ [2,2,1]))\ [n,1]\ ]\!]\ \rho_2\ = \\ \quad \textit{if}\ 2 \le y_1 \le 3\ \textit{then} \\ \quad\quad \langle\{v_1, v_2, v_3\}, \{\{v_1, v_2\}, \{v_2\}\}, \beta, v_{y_1} : v_1, v_1 : v_2 : v_3 - v_{y_1} : v_1\rangle. \end{array} \tag{4}$$

For $y_1 < 2$ or $y_1 > 3$, (4) will be undefined (for an implementation this means that the TDL compiler will print an error message). We continue with computing

$$\begin{array}{ll} \mathcal{N}[\![\ \mathsf{let}\ P\ =\ \mathsf{processor}(\mathtt{type})\ \mathsf{in}\ \mathsf{join}((E)[n,1],\ P)]\!]\ \rho_1 & \\ \quad = \mathcal{N}[\![\mathsf{join}((E)[n,1],\ P)]\!]\ \rho_2 & \text{(with (1) and (4) and renaming)} \\ \quad = \langle\{v_1, v_2, v_3, v_4\}, \{\{v_1, v_2\}, \{v_2\}, \{v_{y_1}, v_4\}\}, \beta', v_{y_1} : v_1 : v_4, v_2 : v_3 - v_{y_1}\rangle & \end{array}$$

where $\beta'(v_i) = y_2$ for $i = 1, 2, 3, 4$.

Finally, we get

$$\mathcal{D}[\![\ \Delta]\!]\ \rho_0\ =\ \rho_0[\mathtt{test}/\theta]$$

where $\theta(y_1, y_2) := \mathcal{N}[\![\mathsf{let}\ P\ =\ \mathsf{processor}(\mathtt{type})\ \mathsf{in}\ \mathsf{join}((E)[n,1],\ P)]\!]\ \rho_1$
$\quad = \langle\{v_1, v_2, v_3, v_4\}, \{\{v_1, v_2\}, \{v_2\}, \{v_{y_1}, v_4\}\}, \beta',\ v_{y_1} : v_1 : v_4,\ v_2 : v_3 - v_{y_1}\rangle.$

Using this, we can compute the semantics of $\mathtt{test}(2,\ \mathsf{HELIOS})$:

$$\begin{array}{l} \mathcal{N}[\![\ \mathtt{test}(2,\ \mathsf{HELIOS})\ ]\!]\ \rho_0[\mathtt{test}/\theta]\ = \\ \quad \langle\{v_1, v_2, v_3, v_4\},\ \{\{v_1, v_2\}, \{v_2\}, \{v_2, v_4\}\},\ \beta'',\ v_2 : v_1 : v_4,\ v_3\rangle, \end{array}$$

where $\beta''(v_i) = \mathsf{HELIOS}$ for $i = 1, 2, 3, 4$.

The resulting graph has the following shape:

We assume that $V_1 \cap V_2 = \emptyset$. Otherwise, the elements of V_2 are renamed appropriately.

Now, we want to define the semantics $\mathcal{D}$ of a set of TDL function definitions. ‡

$\mathcal{D}$: $\underline{\textsc{Def}} \times Env \longrightarrow Env$

$$
\begin{array}{l}
\mathcal{D}[\![\ \mathsf{function}\ \textsc{Id}_1(\textsc{Id}_{1,1},\ldots,\ \textsc{Id}_{1,n_1})\{\textsc{Net}_1\} \\
\qquad \vdots \\
\quad \mathsf{function}\ \textsc{Id}_m(\textsc{Id}_{m,1},\ldots,\ \textsc{Id}_{m,n_m})\{\textsc{Net}_m\}]\!]\ \rho := \rho[\textsc{Id}_1/\theta_1,\ \ldots,\ \textsc{Id}_m/\theta_m] \\
\text{with } \theta_i(y_1,\ldots,y_{n_i}) = \mathcal{N}[\![\ \textsc{Net}_i]\!]\ \rho[\textsc{Id}_1/\theta_1,\ \ldots,\ \textsc{Id}_m/\theta_m,\ \textsc{Id}_{i,1}/y_1,\ldots,\ \textsc{Id}_{i,n_i}/y_{n_i}]
\end{array}
$$

Note that θ_i is defined recursively. $\mathcal{D}$ extends the current environment by new entries for the identifiers of the newly defined functions. Each function identifier is assigned to a function mapping arguments to the graph described by the corresponding function body. This graph is determined using the network semantics $\mathcal{N}$, which we are now going to define.

$\mathcal{N}$: $\underline{\textsc{Net}} \times Env \longrightarrow Graphs$

$$\mathcal{N}[\![\ (\textsc{Net})\ ]\!]\ \rho := \mathcal{N}[\![\ \textsc{Net}\ ]\!]\ \rho$$

$$
\begin{array}{l}
\mathcal{N}[\![\ \mathsf{if}\ \textsc{BExp}\ \mathsf{then}\ \ \textsc{Net}_1\ \mathsf{else}\ \ \textsc{Net}_2]\!]\ \rho := \\
\quad \mathit{if}\ \mathcal{E}[\![\ \textsc{BExp}]\!]\ \rho = \mathit{true}\ \ \mathit{then}\ \mathcal{N}[\![\ \textsc{Net}_1]\!]\ \rho\ \ \mathit{else}\ \mathcal{N}[\![\ \textsc{Net}_2]\!]\ \rho
\end{array}
$$

$$
\begin{array}{l}
\mathcal{N}[\![\ \mathsf{let}\ \textsc{Id}_1 = \textsc{Net}_1,\ \ldots,\ \textsc{Id}_n = \textsc{Net}_n\ \mathsf{in}\ \textsc{Net}\ ]\!]\ \rho := \mathcal{N}[\![\ \textsc{Net}\ ]\!]\ \rho_n \\
\quad \text{where } \rho_1 := \rho[\textsc{Id}_1/\ \mathcal{N}[\![\ \textsc{Net}_1]\!]\ \rho] \\
\qquad\qquad\ \ \rho_2 := \rho_1[\textsc{Id}_2/\ \mathcal{N}[\![\ \textsc{Net}_2]\!]\ \rho_1] \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad\ \ \rho_n := \rho_{n-1}[\textsc{Id}_n/\ \mathcal{N}[\![\ \textsc{Net}_n]\!]\ \rho_{n-1}]
\end{array}
$$

$$\mathcal{N}[\![\ [\textsc{Exp}]\ \textsc{Net}]\!]\ \rho := \quad \mathit{let}\ k := \mathcal{E}[\![\ \textsc{Exp}]\!]\ \rho\ \ \mathit{in}\ \ \mathit{if}\ k \geq 1\ \ \mathit{then} \bigoplus_{i=1}^{k} \mathcal{N}[\![\ \textsc{Net}]\!]\ \rho$$

$$
\begin{array}{l}
\mathcal{N}[\![\ \mathsf{connect}(\textsc{Net},\ \textsc{List}_1,\ \textsc{List}_2)]\!]\ \rho := \\
\quad \mathit{let}\ G := \langle V,E,\beta,L_1,L_2\rangle := \mathcal{N}[\![\ \textsc{Net}\ ]\!]\ \rho, \\
\qquad L' := \mathcal{L}[\![\ \textsc{List}_1]\!]\ G\ \rho, \qquad L'' := \mathcal{L}[\![\ \textsc{List}_2]\!]\ G\ \rho \\
\quad \mathit{in}\ \mathit{if}\ L' \subset L_1\ \ \wedge\ \ L'' \subset L_1\ \ \mathit{then}\ \langle V, E \cup E', \beta, L_1, L_2\rangle \\
\quad \text{where } E' := \mathit{let}\ v_1 \ldots v_n := L',\ \ v'_1 \ldots v'_m := L'',\ \ k := \mathit{min}\{n,m\} \\
\qquad\qquad\qquad \mathit{in}\ \{\{v_1,v'_1\},\ldots,\{v_k,v'_k\}\}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{N}[\![\ \mathsf{join}(\textsc{Net}_1,\ \textsc{Net}_2)]\!]\ \rho := \\
\quad \mathit{let}\ \langle V_i,E_i,\beta_i,L_i,R_i\rangle := \mathcal{N}[\![\ \textsc{Net}_i]\!]\ \rho\quad (\mathit{for}\ i = 1,2), \\
\qquad v_{i,1}\ldots v_{i,n_i} := L_i\ \ (\mathit{for}\ i = 1,2), \qquad k := \mathit{min}\{n_1,n_2\}, \\
\qquad E := \{\{v_{1,1},v_{2,1}\},\ldots,[\{v_{1,k},v_{2,k}\}\} \\
\quad \mathit{in}\ \langle V_1 \cup V_2,\ E_1 \cup E_2 \cup E,\ \beta_1 \odot \beta_2,\ L_1 : L_2,\ R_1 : R_2\rangle
\end{array}
$$

We assume that $V_1 \cap V_2 = \emptyset$. Otherwise, the elements of V_2 are renamed appropriately.

‡notation: $f[x/y](z) := \begin{cases} y & \mathit{if}\ x = z \\ f(z) & \mathit{otherwise} \end{cases}$, $f[x_1/y_1,\ldots,x_n/y_n] := (\ldots(f[x_1/y_1])\ldots)[x_n/y_n]$

$$
\begin{array}{l}
\mathcal{N}[\![\ \mathsf{processor}(\textsc{Attrs})]\!]\ \rho := \langle\{v\},\emptyset,\beta,v,\varepsilon\rangle \\
\quad \text{where } v \text{ is a ``new'' node} \quad \text{and} \quad \beta(v) := \mathcal{E}[\![\ \textsc{Attrs}\ ]\!]\ \rho.
\end{array}
$$

$$
\begin{array}{l}
\mathcal{N}[\![\ \mathsf{node}(\textsc{Net})]\!]\ \rho := \langle\{v\},\emptyset,\beta,v,\varepsilon\rangle \\
\quad \text{where } v \text{ is a ``new'' node} \quad \text{and} \quad \beta(v) := \mathcal{N}[\![\ \textsc{Net}\ ]\!]\ \rho.
\end{array}
$$

$$
\begin{array}{l}
\mathcal{N}[\![\ \textsc{Id}(\textsc{Param}_1,\ldots,\textsc{Param}_n)]\!]\ \rho := \rho(\textsc{Id})(v_1,\ldots,v_n) \\
\quad \text{where } v_i := \begin{cases} \mathcal{E}[\![\textsc{Param}_i]\!]\ \rho & \mathit{if}\ \ \textsc{Param}_i \in \underline{\textsc{Exp}} \cup \underline{\textsc{Attrs}} \\ \mathcal{N}[\![\textsc{Param}_i]\!]\ \rho & \mathit{if}\ \ \textsc{Param}_i \in \underline{\textsc{Net}} \end{cases} \qquad (1 \leq i \leq n)
\end{array}
$$

$$\mathcal{N}[\![\textsc{Id}]\!]\ \rho := \mathit{if}\ \rho(\textsc{Id}) \in \mathit{Graphs}\ \ \mathit{then}\ \ \rho(\textsc{Id})$$

$$
\begin{array}{l}
\mathcal{N}[\![\mathsf{substitute}(\textsc{Net}_1,\textsc{List},\textsc{Net}_2)]\!]\ \rho := \\
\quad \mathit{let}\ G := \langle V,E,\beta,L_1,L_2\rangle := \mathcal{N}[\![\textsc{Net}_1]\!]\ \rho, \qquad L := \mathcal{L}[\![\textsc{List}]\!]\ G\ \rho \\
\quad \mathit{in}\ \mathit{if}\ L \subset L_1\ \ \mathit{then}\ \langle V,E,\beta',L_1,L_2\rangle \\
\quad \text{where } \beta'(v) := \begin{cases} \mathcal{N}[\![\textsc{Net}_2]\!]\ \rho & \mathit{if}\ \ v \in L \\ \beta(v) & \mathit{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{N}[\![\textsc{Net}\ \ \textsc{List}]\!]\ \rho := \\
\quad \mathit{let}\ G := \langle V,E,\beta,L_1,L_2\rangle := \mathcal{N}[\![\textsc{Net}]\!]\ \rho, \qquad L := \mathcal{L}[\![\textsc{List}]\!]\ G\ \rho \\
\quad \mathit{in}\ \mathit{if} \text{ no node occurs more than once in } L\ \mathit{then} \\
\qquad\quad \langle V,E,\beta,L,(L_1 - L):(L_2 - L)\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{N}[\![\mathsf{for}\ \textsc{Id} = \textsc{Exp}_1\ \mathsf{to}\ \textsc{Exp}_2 \\
\qquad \mathsf{next}\ \textsc{Id}_1\ \mathsf{is}\ \textsc{Net}_1, \\
\qquad\qquad \vdots \\
\qquad \mathsf{next}\ \textsc{Id}_n\ \mathsf{is}\ \textsc{Net}_n, \\
\quad \mathsf{result}\ \mathsf{is}\ \textsc{Net}]\!]\ \rho := \\
\quad \mathit{if}\ \ \mathcal{E}[\![\textsc{Exp}_1]\!]\ \rho > \mathcal{E}[\![\textsc{Exp}_2]\!]\ \rho\ \ \mathit{then}\ \mathcal{N}[\![\textsc{Net}]\!]\ \rho \\
\quad \mathit{else}\ \ \mathcal{N}[\![\mathsf{let}\ \textsc{Id}_1 = \textsc{Net}_1, \textsc{Id}_n = \textsc{Net}_n\ \mathsf{in} \\
\qquad\qquad\quad \mathsf{for}\ \textsc{Id} = \textsc{Exp}_1 + 1\ \mathsf{to}\ \textsc{Exp}_2 \\
\qquad\qquad\qquad \mathsf{next}\ \textsc{Id}_1\ \mathsf{is}\ \textsc{Net}_1, \\
\qquad\qquad\qquad\qquad \vdots \\
\qquad\qquad\qquad \mathsf{next}\ \textsc{Id}_n\ \mathsf{is}\ \textsc{Net}_n \\
\qquad\qquad\quad \mathsf{result}\ \mathsf{is}\ \textsc{Net}\ ]\!]\ \rho
\end{array}
$$

The network semantics N uses the scheme $\mathcal{L}$. $\mathcal{L}[\![\ \textsc{List}\ ]\!]\ G\ \rho$ denotes the list of nodes of G, which are specified by $\textsc{List}$ (in the environment ρ):

$\mathcal{L}$: $\underline{\textsc{List}} \times \mathit{Graphs} \times \mathit{Env} \longrightarrow V^*$

$$\mathcal{L}[\![\mathsf{all}]\!]\ \langle V,E,\beta,L_1,L_2\rangle\ \rho := L_1 : L_2$$

$$
\begin{array}{l}
\mathcal{L}[\![[\textsc{Exp},\ \textsc{Sections}]\!]\ G\ \rho := \\
\quad \mathit{let}\ k := \mathcal{E}[\![\textsc{Exp}]\!]\rho, \quad \langle V,E,\beta,v_1:\ldots:v_n,L\rangle\ := G \\
\quad \mathit{in}\ \mathit{if}\ 1 \leq k \leq n\ \mathit{then}\ v_k : (\mathcal{L}[\![[\textsc{Sections}]\!]\ G\ \rho)
\end{array}
$$

is the one given by NET. In general, NET will contain some of the identifiers A_1, A_2, ..., A_n, which are bound to the graphs, which they were bound to after $max\{\text{EXP}_2 - \text{EXP}_1 + 1, 0\}$-times iterating the loop. The syntax of the for-loop respects the declarative nature of the language. By

next A_i is G_i

nothing is modified, but a new object (described by G_i) is defined, which will be bound to the identifier A_i during the next iteration step.

3.9 Miscellaneous

Besides the definition of functions, i.e. parameterized nets, TDL allows to define integer constants and lists of attributes, e.g.

```
const MB = #100000,
      T800_1MB = ptype T800; memory MB; HELIOS;
```

Note that T800_1MB does not denote just a string but a list of attributes. Expressions occuring in such a list are evaluated. In the example, the constant MB is replaced by its meaning, i.e. the hexadecimal constant #100000.

As mentioned, TDL subsumes the Helios RML. Especially, the terminal and subnet definitions may be used. The following is a TDL-description of the topology shown in Figure 1 (possibly using different link numbers):

```
const T800MB = ptype T800; memory #100000; HELIOS;

function matrix(n, m, type){
    if n = 1 then vector(m, type)
    else join(vector(m, type), matrix(n−1, m, type))}

function vector(n, type)
    {connect([n]processor(type), [1..n−1], [2..n])}

subnet /Cluster{
    control Rst_Anl[/Cluster/00];
    terminal 00 {~IO, , matrix(2, 3, T800MB), ; T800MB; Mnode Rst_Anl[pa_ra.d];}
    terminal IO { ; IO}
}
```

4 Formal Semantics of TDL

After the informal explanation of TDL in the last section, we now want to give a formal description of the semantics of TDL. Due to the limited space, we will focus on the semantics of function definitions. A treatment of the remaining features, namely constant, terminal, and subnet definitions, can be found in [St91].

The semantics will be defined by structural induction on the syntax of TDL. For a non-terminal A occuring in the BNF description of the syntax (see Appendix A), $\underline{A}$ will denote the set of all terminal words which may be derived from A.

Since TDL allows to define graphs (networks) and operations on them, we need a convenient representation of graphs. In the following, an (undirected) ***graph*** is a tupel $G := \langle V, E, \beta, L_1, L_2 \rangle$, where

- V is the set of nodes,
- $E \subset \{\{v, v'\} \mid v, v' \in V\}$ is the set of edges,
- $\beta : V \longrightarrow \mathit{Attributes}^* \cup \mathit{Graphs}$
  assignes values to nodes; such a value may be:
  - a list of attributes like the size of memory or the processor type †
  - again a graph (this is needed for the substitute and node constructions); we will not allow that β refers directly or indirectly to G itself.
- $L_1 \in V^*$ is the list of visible nodes,
- $L_2 \in V^*$ is the list of invisible nodes.

We say that a node $v \in V$ ***occurs*** in a list $L \subset V^*$, iff $L = w_1 v w_2$ for some $w_1, w_2 \in V^*$. Each node $v \in V$ must occur exactly once in $L_1 : L_2$ (where ":" denotes the list concatenation). The position of v in $L_1 : L_2$ determines its ***number*** (with respect to G). More precisely, if $v = v_i$ and $v_1 : \ldots : v_n = L_1 : L_2$ (for $v_1, \ldots, v_n \in V$) than v has the number i.

For technical convenience, we will use a canonical generalization of some operations, which are usually defined on sets, to operations on lists, e.g.

$L_1 - L_2$ denotes the list of all elements of L_1 not occuring in L_2,

$v \in L$ tests, whether v occurs in L,

$L_1 \subset L_2$ tests, whether all elements occuring in L_1 also occur in L_2.

Besides graphs, we need the notion of an ***environment***. An environment ρ maps identifiers to constants, graphs, and functions mapping argument values to graphs. ***Env*** denotes the set of all environments.

For the description of the semantics, we need the auxiliary functions $\odot$ and $\oplus$. $\odot$ combines two functions, which are defined on disjoint domains, to a function, which is defined on the union of these domains.

$$\beta_1 \odot \beta_2(v) := \begin{cases} \beta_1(v) & \textit{if } \beta_1(v) \text{ is defined} \\ \beta_2(v) & \textit{otherwise} \end{cases}$$

$\oplus$ combines two graphs without adding connections:

$$\langle V_1, E_1, \beta_1, L_1, R_1 \rangle \oplus \langle V_2, E_2, \beta_2, L_2, R_2 \rangle := \langle V_1 \cup V_2, E_1 \cup E_2, \beta_1 \odot \beta_2, L_1 : L_2, R_1 : R_2 \rangle$$

† $\mathit{Attributes} := \{\text{HELIOS, SYSTEM, IO, NATIVE}\} \cup \{(\text{memory}, z) \mid z \in \mathbb{Z}\} \cup \ldots$

all processors of $\mathbf{net}_1$ are replaced by $\mathbf{net}_2$. The substitution is especially useful to build hierarchical topologies. For people, who prefer building hierarchical topologies in a bottom-up style, the language offers some additional features. As mentioned, functions may have nets as parameters. Furthermore, the node operation allows to see a net as one logical node of a bigger net. With these features, the cube-connected-cycles can be defined as follows:

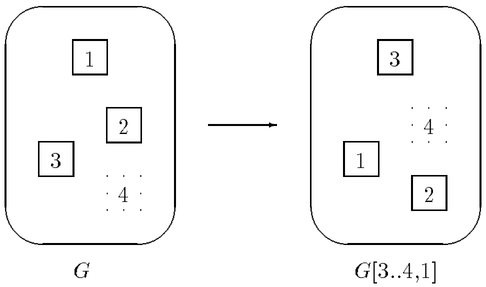
```
function hypercube(dim, net){
    if dim = 0 then node(net)
    else join(hypercube(dim−1, net), hypercube(dim−1, net))}

function ccc(dim, type)
    {hypercube(dim, cycle(dim, type))}
```

3.6 Visibility

Normally, all processors (or nodes) of a net are "visible", i.e. they may be refered to in the processor list of a connect operation or they may be connected with a corresponding processor in a join operation. Sometimes, it is convenient to make processors "invisible". This allows to prohibit that processors are connected to others by a connect or join operation. For instance, if one wants to define a binary tree which may only be connected at the root, this can be done by selecting which processors of a net shall be visible:

```
function tree(height, type){
    if height = 0 then processor(type)
    else (join(join(processor(type),
                    tree(height−1, type)),
               tree(height−1), type))) [1]}
```

In general, **net[list]** causes all processors of **net** which are mentioned in **list** to be visible and the remaining ones to get invisible. Furthermore, the processors are renumbered. The visible processors get the numbers corresponding to their positions in **list**, while the invisible ones get the subsequent numbers. Note that visible processors may get invisible as well as vice versa. The following picture shows an example:



3.7 Local Definitions

To avoid writing the same expression several times, local definitions may be used. An identifier may be bound to an expression and it may be used as an abreviation for it. For instance, a more concise description of the tree above is:

```
function tree(height, type){
    if height = 0 then processor(type)
    else let H = tree(height−1, type)
         in (join(join(processor(type), H), H)) [1]}
```

Note that H is only an abbreviation. Hence, each occurence of H is replaced by a new copy of the corresponding subtree.

3.8 Loops

While many topologies of practical interest can be elegantly composed of smaller ones, some more complicated topologies need to be constructed by repeatedly inserting connections. An example is the shuffle-exchange-topology:

```
function shuffle_exchange(n, type){
    let A = [2 ↑ n]processor(type) in
        for i = 0 to 2 ↑ (n − 1) − 1
            next A is connect(A, [2 ∗ i + 1, i + 1, 2 ↑ (n − 1) + i + 1],
                                 [2 ∗ i + 2, 2 ∗ i + 1, 2 ∗ i + 2])
        result is A}
```

Figure 3 shows a shuffle-exchange-topology with 8 processors. If we assign to processor i $(1 \leq i \leq n)$ the binary representation $b_m \ldots b_1$ of $i-1$, then it is connected to the processors $b_{m-1} \ldots b_1 b_m$ (shuffle) and $b_m \ldots b_2 \bar{b_1}$ (exchange). Like the hypercube, the shuffle-exchange-network has a logarithmic diameter.

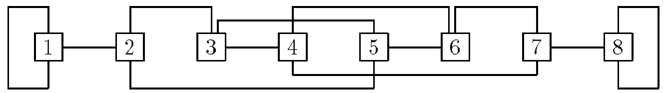


Figure 3: shuffle-exchange-network with 8 processors; horizontal lines represent exchange connections, the others shuffle connections.

The graph described by a for-loop

```
for I = EXP_1 to EXP_2
    next A_1 is G_1
    next A_2 is G_2
        ⋮
    next A_n is G_n
result is NET
```

approach is more natural. Additionally, it has the advantage that it fits well into RML, which is also declarative.

3.1 Single Processors and Replication

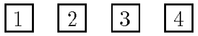
The simplest net just consists of one processor:

processor(**type**)

where **type**, as in RML, may be an arbitrary sequence of attributes (see Appendix A).

Using replication, a net consisting of several identical subnets may be defined, e.g.:

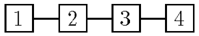
[4] processor(**type**)



3.2 Adding Connections

The processors of a network are internally numbered. Using these numbers, the basic operation connect allows to add connections to a net. For instance, a vector of four processors can be defined by:

connect([4]processor(**type**),[1,2,3],[2,3,4])



connect takes three arguments: a net and two lists of processor numbers. It produces a net like that given as the first argument, but where additionally the processors at corresponding list positions are connected. If one list is longer than the other, the surplus elements are ignored. If a software-configurable multiprocessor system is used, there is no need to talk about the numbers of physical links. The configuration software can establish the desired topology independently of the link numbers selected. In fact, if two processors are to be connected, the TDL compiler will use the first free link numbers of the processors, but this is transparent to the user. A program, i.e. a task force, is anyway independent of the physical link numbers, since the routing of messages is handled by the operating system. A slightly modified version of connect, which allows to specify the link numbers, is discussed in Section 5.

3.3 Functions

Since a vector is a structure, which may be used to build larger nets, it is convenient to assign a name to it:

```
function vector(n, type)
    {connect([n]processor(type), [1..n−1], [2..n])}
```

Additionally, this allows to define parameterized nets. In the example, the length of the **vector** and the **type** of each processor are given as parameters. Besides an integer and a list of attributes, an argument may be a whole net.

3.4 Recursion, Conditional, and Join

Using the above **vector**, we can now easily define a **matrix**:

```
function matrix(n, m, type){
    if n = 1 then vector(m, type)
    else join(vector(m, type), matrix(n−1, m, type))}
```

This example shows some additional features of TDL: recursion, conditional, and join. A $n \times m$-matrix consists of a vector and a $(n-1) \times m$-matrix. Hence, recursion allows to define a matrix very elegantly. A conditional is needed to handle the base case ($n = 1$) separately. The basic operation join combines two nets by connecting equally numbered processors. After this, the processors (of the second net) are renumbered as shown in Figure 2.

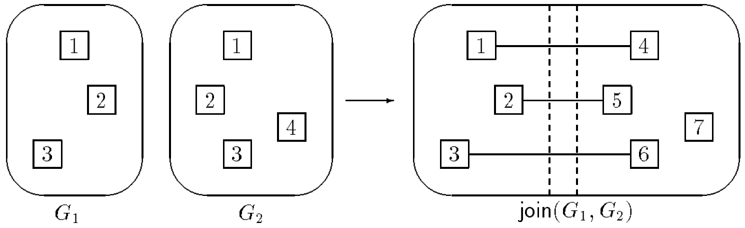


Figure 2: joining two graphs

While a matrix can also be described by a loop, some topologies have a recursive structure and are hence most naturally described by recursion. A good example is the hypercube:

```
function hypercube(dim, type){
    if dim = 0 then processor(type)
    else join(hypercube(dim−1, type),
              hypercube(dim−1, type))}
```

3.5 Substitution

Unfortunately, the number of links of a processor is limited (four in the case of transputers). Hence, hypercubes of arbitrary dimension cannot be constructed. In order to build large hypercube-like topologies, each node of the hypercube must be replaced by a small subnet. If the nodes are replaced by cycles, we get the well-known cube-connected-cycles, which can be described as follows:

```
function ccc(dim, type)
    {substitute(hypercube(dim,type), all, cycle(dim, type))}

function cycle(n, type)
    {connect(vector(n, type), [1], [n])}
```

In general, substitute(**net**$_1$, **list**, **net**$_2$) replaces all processors of **net**$_1$ which are mentioned in **list** by (new copies of) **net**$_2$. If the keyword all is given instead of a processor list,

2 Helios

Helios is a UNIX-like distributed operating system designed for transputer based architectures. Each processor runs a nucleus which controls its own resources and integrates it into the global system. Helios is based on the client-server-model. If a client wants some service (e.g. file I/O), it sends a message to a server, which executes the operation and returns a reply message. In order to achieve fault tolerance, all servers are stateless. All the needed information is contained in the request message.

Helios requires a description of the hardware. For this purpose, it provides a so-called resource map language. In this language, the processors are declared one after the other. For each processor, it is specified to which other processors it is directly connected and which attributes it has (e.g. size of memory, processor type, ...). The following example resource map defines a 2×3 matrix of T800 transputers connected to a root transputer (00). Each transputer has 1 MB of memory and runs Helios. Moreover, the root transputer is connected to the host processor. The control and mnode specifications concern Helios details and we will not explain them here.

```
subnet /Cluster{
control Rst_Anl [/Cluster/00]
terminal 00 {~IO,      ,~01 ,      ; ptype T800; HELIOS; memory #100000;
                                     mnode Rst_Anl[pa_ra.d];}
terminal 01 {~00,      ,~02 ,~04 ; ptype T800; HELIOS; memory #100000;}
terminal 02 {~01,      ,~03 ,~05 ; ptype T800; HELIOS; memory #100000;}
terminal 03 {~02,      ,     ,~06 ; ptype T800; HELIOS; memory #100000;}
terminal 04 {      ,~01 ,~05 ,      ; ptype T800; HELIOS; memory #100000;}
terminal 05 {~04,~02 ,~06 ,      ; ptype T800; HELIOS; memory #100000;}
terminal 06 {~05,~03 ,     ,      ; ptype T800; HELIOS; memory #100000;}
terminal IO {                        ; IO;}
```

The resulting network is shown in Figure 1.

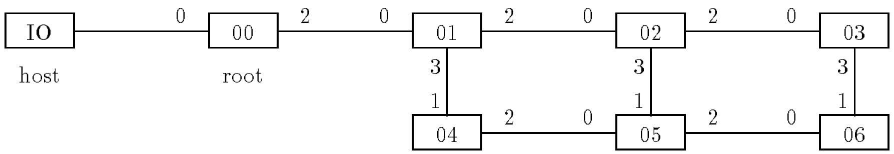


Figure 1: 2×3-matrix connected to a root and a host processor. The numbers 0,1,2, and 3 indicate, which physical links have been used. They correspond to the positions in the argument lists of the terminal declarations.

For a more detailed explanation of RML and Helios in general refer to [Pe89a][Pe89b].

After Helios is booted, the user may run several ***task forces***. A task force consists of one or more cooperating ***tasks***, each of which is automatically mapped onto one processor. If the number of tasks exceeds the number of processors, several tasks are mapped onto a single processor. The tasks communicate via messages. The multiplexing and routing of messages

is done by Helios and transparent to the user. Each task consists of one or more ***processes***, which are scheduled by the transputer.

The structure of a task force is described using a so-called component distribution language (CDL). This language allows to specify some tasks which communicate via unidirectional or bidirectional (UNIX-like) pipes. Pipes may be named in order to facilitate refering to them. Replication is offered as a control structure, which allows to easily specify regularly structured task forces with many components.

The following CDL-script specifies a 2×3-torus of tasks which all communicate with a controller component via bidirectional pipes (% denotes the modulo operation, < | is followed by an input pipe, > | by an output pipe).

```
component element[i,j]{
   streams , , , ,
            < | right{i,j},          > | right{i, (j+1) % 3},
            < | left{i, (j+1) % 3}, > | left{i, j},
            < | down{i,j},           > | down{(i+1) % 2, j},
            < | up{(i+1) % 2,j},    > | right{i, j},
;}
controller (,[i<2, j<3] <> (element{i,j}))
```

Helios automatically maps each task to a processor. This static mapping is based on a heuristic, which takes the hardware requirements (such as processor type, size of memory) into account and tries to place communicating tasks as close to each other as possible. The user can guide the heuristic by explicitely specifying the processor number for some task. Furthermore, it is possible to assign some attributes to certain processors (in the resource map) and to require that certain tasks are only mapped to processors having these attributes. A task which has been assigned to a processor cannot be moved to another one.

3 The Topology Description Language TDL

While the Helios RML and a graphical tool are only suited for describing small topologies, the occam configuration sublanguage as well as CDL (which of course describes software topologies) are lacking some structuring facilities like functions and procedures known from programming languages. Although it is possible to describe every topology, this may be tedious, hard to modify, and, most of all, hard to understand, since the connection structure is not obvious but coded into the placement of (named) channels/pipes.

In this section, we are going to present the topology description language TDL, which offers the structuring facilities known from programming languages (e.g. functions, replication, if-then-else, recursion, loops) as well as a set of basic operations custom tailored to the description of topologies.

In order to guarantee upward compatibility with RML, TDL is just a superset of RML.

When designing a topology description language, there are essentially two alternatives. One can either take an imperative language, where some state is transformed, until the desired topology is produced, or one can use a declarative language, where the topology is just described. Since a network topology is something static, the second (and selected)

The Topology Description Language TDL

Herbert Kuchen, Holger Stoltze
RWTH Aachen*

Friedrich Lücking
Parsytec Computer GmbH†

Abstract

We present a declarative language, called TDL, which allows to describe the topology of a loosely coupled multiprocessor system elegantly, clearly, and concisely. TDL includes some concepts known from programming languages such as loops, conditionals, functions, recursion, and replication. Moreover, TDL offers a set of basic operations suited for the generation of graphs. The TDL compiler transforms a TDL script into a format, which is accepted by the boot program of the distributed operating system Helios running on transputer systems.

Although, TDL has been designed for the description of topologies of multiprocessor systems, it is essentially a graph description language, which can be used in many other contexts.

We give an informal explanation of TDL as well as a formal description of its syntax and semantics.

1 Introduction

In the last years, ***loosely coupled multiprocessor systems*** have become more and more widespread. Such a system consists of several processors which only have local memory and which communicate via message passing. These messages are send along buses or point-to-point connections, so-called ***links***. Here, we are only interested in systems using links. For technical reasons, the number of links per processor is limited by a small constant, e.g. 4 for transputer systems [IN88a]. Hence, each processor is only connected to a small number of neighbours. Messages to the other processors have to be routed.

The structure of such a multiprocessor system forms a ***graph***, where the processors are the nodes and the links are the edges. In the following, we will also use the terms ***topology*** and ***network*** for such graphs.

There are multiprocessor systems where the topology is fixed (e.g. the Intel iPSC[1] [TW91] and some transputer systems) and such where the topology can be established by some configuration software before running a program (e.g. the Parsytec Supercluster [KL88], based on transputers).

*RWTH Aachen, Lehrstuhl für Informatik II, Ahornstraße 55, D-5100 Aachen, Germany, email: herbert@zeus.informatik.rwth-aachen.de , holger@zeus.informatik.rwth-aachen.de

†Parsytec Computer GmbH, Jülicher Str. 338, D-5100 Aachen, Germany, email: fritz@parsytec.de

[1]In this paper, we mention the following registered trademarks: Helios (of Perihelion Software Limited, iPSC (of Intel Corporation), Supercluster (of Parsytec GmbH), occam and transputer (of the INMOS group of companies), and UNIX (of AT&T).

There are essentially two ways, how such a multiprocessor system can be used. The first approach is that the user renounces an operating system and that he explicitely maps each task to a processor. This allows to exploit the knowledge on the expected communication behaviour in order to place havily communicating tasks physically close to each other. The software topology, i.e. the way how the cooperating tasks communicate, depends on the hardware topology, i.e. the way how the processors are connected. In general, changing the hardware topology implies changing the software. Programming comfort and flexibility are traded for performance. This approach is supported by occam [IN88b] and ParC [Pa90] on transputer systems. The network description language (NDL) of the new Inmos Toolset is a bit more flexible. It allows to map several tasks to each processor. The mapping of software channels to physical links is done automatically.

The second approach is to use a distributed operating system like Helios [Pe89a] which automatically maps tasks to processors. Here, the software topology is independent of the hardware topology. The routing and multiplexing of messages are handled by the operating system.

In any case, a description of the hardware topology is needed. If a software-configurable multiprocessor system is used, the description is needed by the configuration software to establish the desired topology. Moreover, the description is needed to boot the system.

In the case of Helios, the operating system itself needs a description of the hardware. The Helios hardware description language, called resource map language (RML), requires a separate declaration of each processor together with its physical links and attributes.

In the case of occam, hardware and software topology are in fact identical and they are described by the configuration sublanguage of occam. This language allows to map one (possibly structured) process to each processor and communication channels to physical links. The only control structure of this sublanguage is replication. The Toolset NDL additionally provides a conditional (and the possibility to map several tasks to each processor).

Such a language is only appropriate for systems with a small number of processors or for systems with a very special structure. For instance, the ubiquitious hypercube and hierarchical networks are hard to specify. Moreover, using a language like RML, the structure of the network is obscured, since it is coded into the placement of channels/pipes.

An alternative to a textual topology description is to use a graphical design tool such as Gracia [Kr88]. Using a mouse, processors can be defined and connected. This approach is fine for a small number of processors. For hundreds or thousands of processors, the screen is not adequate. A hierachical solution could help, but this would restrict the possible topologies.

In this paper, we will propose the new topology decription language TDL, which includes many concepts known from programming languages. Although, it was developed as a language to describe hardware topologies, the approach is much more general. In fact, TDL can be seen as a general graph description language.

This paper is organized as follows. In Section 2, we will shortly describe the software and hardware environment of Helios. Section 3 contains an informal description of TDL. In Section 4, we give the formal semantics of TDL. In Section 5, we conclude and point out some extensions of the language.