

SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-42

**Refining the Dependency
Management System
using the Refinement Calculus**

David Carrington and Nigel Ward

March 1995

Phone: +61 7 365 1003

Fax: +61 7 365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

Refining the Dependency Management System using the Refinement Calculus

David Carrington and Nigel Ward

Abstract

This report shows how a specification of the Dependency Management System can be refined towards an implementation using the refinement calculus.

1 Introduction

The Dependency Management System is a case study developed within the Software Verification Research Centre. An introduction to the case study is provided by Peter Lindsay[6]. This report demonstrates how a module representing the Dependency Management System can be refined. The refinement uses the refinement calculus [7, 10, 1] emphasising data refinement towards an efficient implementation.

2 Introduction to the refinement calculus

The refinement calculus is based on the addition of specification constructs to Dijkstra's guarded command language [3]. Because the refinement calculus has been discovered a number of times, it has a number of different notational flavours [7, 10, 1]. We use a notation close to that described in [7].¹

¹One difference is our **invariant** clause in a module.

The executable components of the language include:

- skip and abort,
- assignment,
- sequential composition,
- alternation,
- repetition,
- local variable blocks,
- parameterised recursive procedures,
and
- modules.

One of the language's (possibly non-executable) specification constructs is the *specification statement*:

$$w: [pre, post].$$

When executed in a state satisfying the predicate *pre* this statement terminates in a state satisfying the predicate *post* by only changing the values of the variables *w*. Zero subscripted variables in *post* refer to the values of those variables in the initial state.

Specifications are usually written in the refinement calculus as a mixture of executable statements and specification statements. For example, consider a system which maintains a set which is limited to N elements. Operations on the set allow the addition, selection and removal of nodes. Initially the set of nodes is empty. This system could be specified as in Figure 1. Note that the *Add* operation has $\#elements < N$ as part of its pre-condition to ensure that the module invariant ($\#elements \leq N$) is maintained. The other operations trivially maintain it.

By using specifications which are a mixture of executable and specification constructs, the refinement calculus blurs the distinction between specification and program and the two terms can be used interchangeably. Once a specification has been obtained, the refinement calculus gives the developer a collection of refinement laws which can be applied to the specification to transform it into executable code while guaranteeing correctness. These laws change either the algorithmic structure (procedural refinement) or the data representation (data refinement) of the specification.

```

module BoundedSet  $\hat{=}$ 
  var elements :  $\mathbb{P} X$ ;
  invariant  $\#elements \leq N$ ;
  initially elements =  $\emptyset$ ;
  procedure Add(value x : X)  $\hat{=}$ 
    elements :  $\left[ \begin{array}{l} x \notin elements \\ \#elements < N \end{array} , elements = elements_0 \cup \{x\} \right]$ ;
  procedure Select(result x : X)  $\hat{=}$ 
    x :  $[elements \neq \emptyset , x \in elements]$ ;
  procedure Remove(value x : X)  $\hat{=}$ 
    elements :  $[x \in elements , elements = elements_0 \setminus \{x\}]$ ; end

```

Figure 1: Bounded set module

2.1 Procedural refinement

Procedural refinement involves the application of refinement laws which change the algorithmic structure of a specification. For example, $n : [x \geq 0 , n = x!]$ can be refined to

```

if  $x = 0 \rightarrow n : [x = 0 , n = x!]$ 
 $\parallel$   $x > 0 \rightarrow n : [x > 0 , n = x!]$ 
fi

```

using the following refinement law (and some logical simplification):

Alternation If $pre \Rightarrow (G_1 \vee G_2)$ then

$$w : [pre , post] \sqsubseteq \begin{array}{l} \mathbf{if} \ G_1 \rightarrow w : [G_1 \wedge pre , post] \\ \parallel \ G_2 \rightarrow w : [G_2 \wedge pre , post] \\ \mathbf{fi} \end{array}$$

Refinement laws often have side conditions (such as $pre \Rightarrow (G_1 \vee G_2)$ above) which must be proven for the refinement to be correct.

2.2 Data refinement

Data refinement is a refinement that replaces an “abstract” data type in a module by a “concrete” data type, i.e.

<pre> module $AM \cong$ var $a : A$ invariant Inv initially $Init$ procedure $P \cong S$ end </pre>	<p>is refined to</p>	<pre> module $CM \cong$ var $c : C$ invariant Inv' initially $Init'$ procedure $P \cong S'$ end </pre>
--	------------------------------	---

where Inv , $Init$ and S (the body of P), all referring to variables a , are refined to reference variables c . The algorithmic structure of S is preserved by this transformation. The transformation requires an abstraction invariant AI which links the abstract variables a with the concrete variables c .

We use an approach that calculates the new invariant Inv' , initialisation $Init'$ and the procedure body S' as a consequence of the choice of the abstraction invariant. The concrete invariant Inv' must satisfy $Inv' \Rightarrow \exists a : A \bullet AI \wedge Inv$. The concrete initialisation $Init'$ must satisfy $Init' \Rightarrow \exists a : A \bullet AI \wedge Init$. In S , we replace $a, x : [Pre, Post]$ with

$$c, x : [\exists a : A \bullet AI \wedge Pre, \exists a : A \bullet AI \wedge Post]$$

Where the abstraction invariant AI is functional (i.e. at most one abstract value for each concrete value), we can express it as follows:

$$AI \cong a = AF(c) \wedge CI$$

where AF is the abstraction function linking abstract and concrete state (a and c respectively), and CI is a concrete state invariant. For this situation, we get simpler calculations:

$a, x : [Pre, Post]$ becomes $c, x : [Pre_{[a \setminus AF(c)]}, Post_{[a \setminus AF(c)}]]$

i.e. simple substitution² for a with $AF(c)$. Further details can be found in [7, 4, 8, 2, 9].

3 Initial specification

The initial specification is shown in Figure 2. This specification was derived systematically from the Object-Z specification developed by Gordon Rose using techniques similar to those described in [5]. Only the five selected operations are shown.

3.1 State

The state of an Object-Z system translates fairly directly into the refinement calculus. The variables declared in the state schema become the variables of the refinement calculus module, the state invariant becomes the module invariant, and, in our case, the state functions become simple syntactic abbreviations.

$nodes : \mathbb{F} X$ $dir_dep_on : X \leftrightarrow X$ \rightarrow $_ \succ _ : X \leftrightarrow X$
$dir_dep_on \subseteq nodes \times nodes$ $\succ = dir_dep_on^+$ $\#x : X \bullet x \succ x$

²The notation $P_{[a \setminus b]}$ is the formula P with all free occurrences of a replaced by b .

```

module DepManSys  $\hat{=}$ 
  var nodes :  $\mathbb{F} X$ ;
      dir_dep_on :  $X \leftrightarrow X$ ;

  abbreviations
     $\succ \hat{=} \text{dir\_dep\_on}^+$ ,     $\succeq \hat{=} \text{dir\_dep\_on}^*$ 

  invariant
     $\text{dir\_dep\_on} \subseteq \text{nodes} \times \text{nodes} \wedge \nexists x : X \bullet x \succ x$ 

  initially
     $\text{nodes} = \emptyset \wedge \text{dir\_dep\_on} = \emptyset$ 

  procedure RemoveNode(value  $x : X$ )  $\hat{=}$ 
     $\text{nodes}, \text{dir\_dep\_on} : \left[ \begin{array}{l} x \in \text{nodes} \setminus \text{nodes}_0 \setminus \{x\} \\ \text{ran } \text{dir\_dep\_on} \setminus \{x\} \triangleleft \text{dir\_dep\_on}_0 \end{array} \right];$ 

  procedure CanAdd(value  $x, y : X$ ; result  $b : \mathbb{B}$ )  $\hat{=}$ 
     $b : [\{x, y\} \subseteq \text{nodes} \ , \ b \Leftrightarrow (y \not\succeq x)]$ ;

  procedure AddDependence(value  $x, y : X$ )  $\hat{=}$ 
     $\text{dir\_dep\_on} : \left[ \begin{array}{l} \{x, y\} \subseteq \text{nodes} \\ y \not\succeq x \end{array} \ , \ \text{dir\_dep\_on} = \text{dir\_dep\_on}_0 \cup \{(x, y)\} \right];$ 

  procedure Supporters(value  $x : X$ ; result  $\text{nds} : \mathbb{F} X$ )  $\hat{=}$ 
     $\text{nds} : [x \in \text{nodes} \ , \ \text{nds} = \{n : \text{nodes} \mid x \succ n\}]$ ;

  procedure SomeDirectDependent(value  $x : X$ ; result  $\text{node} : X$ )  $\hat{=}$ 
     $\text{node} : [x \in \text{ran } \text{dir\_dep\_on} \ , \ (\text{node}, x) \in \text{dir\_dep\_on}]$ 

end

```

Figure 2: Initial module specification

becomes

```

var
  nodes :  $\mathbb{F} X$ ;
  dir_dep_on :  $X \leftrightarrow X$ ;
abbreviations
   $\succ \hat{=} dir\_dep\_on^+$ ;
   $\succcurlyeq \hat{=} dir\_dep\_on^*$ ;
invariant
   $dir\_dep\_on \subseteq nodes \times nodes$ 
   $\#x : X \bullet x \succ x$ 

```

Note that we have introduced another state function, \succcurlyeq . This simplifies the specification of the *AddDependence* operation.

3.2 Initialisation

The module initialisation is derived from the Object-Z specification by conjoining the predicate of the *INIT* schema with the module invariant and then simplifying.

$$\begin{aligned}
 nodes &= \emptyset \wedge dir_dep_on \subseteq node \times node \wedge (\#x : X \bullet x \succ x) \\
 &\equiv node = \emptyset \wedge dir_dep_on = \emptyset \wedge (\#x : X \bullet (x, x) \in dir_dep_on^+) \\
 &\equiv nodes = \emptyset \wedge dir_dep_on = \emptyset
 \end{aligned}$$

3.3 Operations

Object-Z and the refinement calculus use different conventions for naming variables in the “pre” and “post” states. The first task in the translation of operations is to convert the undashed / dashed convention of Object-Z to the zero-subscript convention of the refinement calculus. If Op is an Object-Z operation then we let \overline{Op} denote the same operation using the refinement calculus convention.

An Object-Z operation of the form

\overline{Op}
$\Delta(vars)$
$i? : I$
$o! : O$
$pred$

can be translated to a refinement calculus procedure definition of the form

procedure $Op(\text{value } i : I; \text{result } o : O) \hat{=} w : [(\exists w \mid inv \bullet pred)_{[w_0 \setminus w][i?, o! \setminus i, o]}, pred_{[i?, o! \setminus i, o]}]$

where

- the frame w consists of the delta variables $vars$ and the output variables o ,
- inv is the state invariant found in the module **invariant** section,
- the renaming $_{[w_0 \setminus w]}$ ensures that the pre-condition is expressed only in terms of undecorated variables, and
- the renaming $_{[i?, o! \setminus i, o]}$ removes the decorations from the input and output variables.

To illustrate this translation, we apply it to the Object-Z *RemoveNode* specification. Firstly, we change from the Object-Z to the refinement calculus naming convention.

$\overline{RemoveNode}$
$\Delta(nodes, dir_dep_on)$
$x? : X$
$x? \in nodes_0 \setminus \text{ran } dir_dep_on_0$
$nodes = nodes_0 \setminus \{x?\}$
$dir_dep_on = \{x?\} \triangleleft dir_dep_on_0$

Calculating the pre-condition of the corresponding refinement calculus specification statement gives

$$\begin{aligned}
& (\exists \text{ nodes}, \text{ dir_dep_on} \mid \text{ dir_dep_on} \subseteq \text{ nodes} \times \text{ nodes} \wedge (\nexists x : X \bullet x \succ x) \bullet \\
& \quad x? \in \text{ nodes}_0 \setminus \text{ ran } \text{ dir_dep_on}_0 \\
& \quad \text{ nodes} = \text{ nodes}_0 \setminus \{x?\} \\
& \quad \text{ dir_dep_on} = \{x?\} \triangleleft \text{ dir_dep_on}_0 \\
&)_{[\text{ nodes}_0, \text{ dir_dep_on}_0 \setminus \text{ nodes}, \text{ dir_dep_on}][x? \setminus x]} \\
& \equiv x \in \text{ nodes} \setminus \text{ ran } \text{ dir_dep_on}
\end{aligned}$$

Using this, *RemoveNode* translates to

$$\begin{aligned}
& \text{procedure } \text{RemoveNode}(\text{value } x : X) \hat{=} \\
& \quad \text{ nodes}, \\
& \quad \text{ dir_dep_on} \dot{=} \left[\begin{array}{l} x \in \text{ nodes} \setminus \\ \text{ ran } \text{ dir_dep_on} \end{array} , \begin{array}{l} x \in \text{ nodes}_0 \setminus \text{ ran } \text{ dir_dep_on}_0 \\ \text{ nodes} = \text{ nodes}_0 \setminus \{x\} \\ \text{ dir_dep_on} = \{x\} \triangleleft \text{ dir_dep_on}_0 \end{array} \right]
\end{aligned}$$

The refinement calculus lets us assume the pre-condition of a specification statement in its post-condition so this specification statement can be simplified.

$$\begin{aligned}
& \text{ nodes}, \\
& \text{ dir_dep_on} \dot{=} \left[\begin{array}{l} x \in \text{ nodes} \setminus \\ \text{ ran } \text{ dir_dep_on} \end{array} , \begin{array}{l} \text{ nodes} = \text{ nodes}_0 \setminus \{x\} \\ \text{ dir_dep_on} = \{x\} \triangleleft \text{ dir_dep_on}_0 \end{array} \right]
\end{aligned}$$

The other operations in the refinement calculus specification can be derived similarly. Operations that compute a Boolean result (such as *CanAdd*) need an explicit **result** parameter added to the procedure declaration and to the frame of the specification statement. The post-condition becomes $b \Leftrightarrow \text{post}$ where b is the result parameter.

4 New data representation — step 1

The initial design step involves data refinement of the *DepManSys* module. The transformation changes the representation of the *dir_dep_on* relation while leaving the *nodes* set unchanged. The new concrete state is the original *nodes* set plus two functions:

$$\begin{aligned}
& \text{deps} : X \rightarrow \mathbb{F} X \\
& \text{sups} : X \rightarrow \mathbb{F} X
\end{aligned}$$

These functions are related to dir_dep_on as follows:

$$\begin{aligned}
deps\ x &= \text{“the set of nodes that node } x \text{ directly depends on”} \\
&= dir_dep_on(\{x\}) \\
sups\ x &= \text{“the set of nodes that node } x \text{ directly supports”} \\
&= dir_dep_on^{\sim}(\{x\})
\end{aligned}$$

These two functions introduce some redundancy since either function may be derived from the other but the redundancy improves the efficiency of the module. As such it is a deliberate design decision. In terms of the original specification, the domain and range restriction of the dir_dep_on relation for all singleton subsets of $nodes$ is stored in $deps$ and $sups$.

$$\begin{aligned}
ran(\{x\} \triangleleft dir_dep_on) &= deps\ x \\
dom(dir_dep_on \triangleright \{x\}) &= sups\ x
\end{aligned}$$

This choice of data representation simplifies access to the information in the relation by partitioning it. Study of the DMS operations reveals that the relation needs to be accessed both through its domain and range elements.

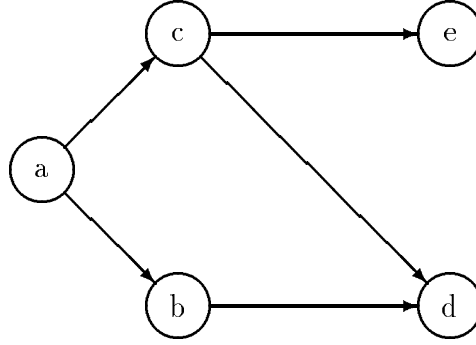
Figure 3 illustrates the relationship between the relation dir_dep_on and the $deps$ and $sups$ functions.

The goal is a further data refinement step leading to a representation that combines $nodes$, $deps$ and $sups$ in a function $nodemap : X \rightarrow (\mathbb{F} X \times \mathbb{F} X)$. This could be implemented as a table indexed by nodes that gives both direct dependents and direct supporters for each node.

4.1 Abstraction invariant

The relationship between the original abstract state (dir_dep_on) and the new concrete state ($deps$ and $sups$) is crucial to the refinement process since it is used to calculate the specification of the each operation in terms of the new state. The calculation of the concrete version of each operation is simplified because the relationship is functional, i.e. each concrete state maps to a single abstract state.

The auxiliary function mk_rel is used to define the abstraction invariant.



$$\begin{aligned}
dir_dep_on &= \{a \mapsto b, a \mapsto c, b \mapsto d, c \mapsto d, c \mapsto e\} \\
deps &= \{a \mapsto \{b, c\}, b \mapsto \{d\}, c \mapsto \{d, e\}, d \mapsto \{\}, e \mapsto \{\}\} \\
sups &= \{a \mapsto \{\}, b \mapsto \{a\}, c \mapsto \{a\}, d \mapsto \{b, c\}, e \mapsto \{c\}\}
\end{aligned}$$

Figure 3: Dependency relation example

$mk_rel : (X \mapsto \mathbb{F} X) \rightarrow (X \leftrightarrow X)$
$\forall f : X \mapsto \mathbb{F} X \bullet$ $mk_rel f = \{a, b : X \mid a \in \text{dom} f \wedge b \in f a \bullet a \mapsto b\}$

As the abstraction invariant is functional, it can be written as an abstraction function AF and a concrete invariant CI . The original module invariant can be assumed to simplify the relationship between the abstract and concrete states.

$$\begin{aligned}
a = AF(c) &\hat{=} dir_dep_on = mk_rel\ deps \\
&\equiv dir_dep_on = \{n1, n2 : nodes \mid n2 \in deps\ n1 \bullet n1 \mapsto n2\}
\end{aligned}$$

The new concrete invariant is:

$$\begin{aligned}
CI &\hat{=} (\text{dom } \textit{deps}) = \textit{nodes} \wedge (\text{dom } \textit{sups}) = \textit{nodes} \\
&\wedge \bigcup \text{ran } \textit{deps} \subseteq \textit{nodes} \wedge \bigcup \text{ran } \textit{sups} \subseteq \textit{nodes} \\
&\wedge \textit{mk_rel } \textit{deps} = (\textit{mk_rel } \textit{sups})^\sim \\
&\wedge (\textit{mk_rel } \textit{deps})^+ \cap \text{id } X = \emptyset
\end{aligned}$$

From the abstraction function and the concrete invariant, some additional relationships can be derived:

$$\begin{aligned}
\textit{dir_dep_on} &= (\textit{mk_rel } \textit{sups})^\sim \\
&= \{n1, n2 : \textit{nodes} \mid n1 \in \textit{sups } n2\}
\end{aligned}$$

and hence

$$\forall n1, n2 : \textit{node} \bullet n1 \in \textit{sups } n2 \Leftrightarrow n2 \in \textit{deps } n1$$

4.2 New concrete module

The new version of the DMS module is calculated as a consequence of the abstraction invariant decision.

4.2.1 initial state

The new concrete initial state is given by the formula $I_{[a \setminus AF(c)]} \wedge CI$ where a is the abstract state and I is the original initialisation. So substituting for $\textit{dir_dep_on}$ in $\textit{nodes} = \emptyset \wedge \textit{dir_dep_on} = \emptyset$ and adding CI gives

$$\begin{aligned}
&\textit{nodes} = \emptyset \wedge \textit{mk_rel } \textit{deps} = \emptyset \wedge \\
&(\text{dom } \textit{deps}) = \textit{nodes} \wedge (\text{dom } \textit{sups}) = \textit{nodes} \wedge \\
&\bigcup \text{ran } \textit{deps} \subseteq \textit{nodes} \wedge \bigcup \text{ran } \textit{sups} \subseteq \textit{nodes} \wedge \\
&\textit{mk_rel } \textit{deps} = (\textit{mk_rel } \textit{sups})^\sim \wedge \\
&(\textit{mk_rel } \textit{deps})^+ \cap \text{id } X = \emptyset
\end{aligned}$$

which can be simplified (using $\textit{nodes} = \emptyset$) to

$$\textit{nodes} = \emptyset \wedge \textit{deps} = \emptyset \wedge \textit{sups} = \emptyset$$

4.2.2 concrete operations

For each operation specified as $a, x \bullet [Pre, Post]$ the computed specification on the concrete state is

$$c, x \bullet [Pre_{[a \setminus AF(c)]}, Post_{[a \setminus AF(c)]}]$$

The concrete invariant is implicitly conjoined to the pre-condition and the post-condition of each operation.

RemoveNode The original pre-condition is transformed

$$\begin{aligned} x \in (nodes \setminus \text{ran } dir_dep_on) \\ &\equiv x \in nodes \wedge x \notin \text{ran } dir_dep_on \\ &\equiv x \in nodes \wedge \neg (\exists n : nodes \bullet x \in deps\ n) \\ &\equiv x \in nodes \wedge \neg (\exists n : nodes \bullet n \in sups\ x) \\ &\equiv x \in nodes \wedge sups\ x = \emptyset \end{aligned}$$

The transformation of the original post-condition

$$nodes = nodes_0 \setminus \{x\} \wedge dir_dep_on = \{x\} \triangleleft dir_dep_on_0$$

is more complicated. In this case it is more convenient to compute the change in *deps* and *sups* separately based on the change in *dir_dep_on*.

$$\begin{aligned} deps &= \{n : nodes \bullet n \mapsto dir_dep_on(| \{n\} |)\} \\ &= \{n : nodes_0 \setminus \{x\} \bullet n \mapsto (\{x\} \triangleleft dir_dep_on_0)(| \{n\} |)\} \\ &= \{n : nodes_0 \setminus \{x\} \bullet n \mapsto deps_0\ n\} \\ &= \{x\} \triangleleft deps_0 \end{aligned}$$

The equivalent calculation for *sups* is more involved.

$$\begin{aligned} sups &= \{n : nodes \bullet n \mapsto dir_dep_on^{\sim}(| \{n\} |)\} \\ &= \{n : nodes_0 \setminus \{x\} \bullet n \mapsto (\{x\} \triangleleft dir_dep_on_0)^{\sim}(| \{n\} |)\} \\ &= \{n : nodes_0 \setminus \{x\} \bullet n \mapsto dir_dep_on_0^{\sim}(| \{n\} |) \setminus \{x\}\} \\ &= \{n : nodes_0 \setminus \{x\} \bullet n \mapsto sups_0\ n \setminus \{x\}\} \end{aligned}$$

This expression could be the post-condition, but it is helpful to continue the transformation to indicate an efficient implementation.

$$\begin{aligned}
&= \{n : nodes_0 \setminus (\{x\} \cup deps_0 x) \bullet n \mapsto sups_0 n \setminus \{x\}\} \\
&\quad \cup \{n : deps_0 x \bullet n \mapsto sups_0 n \setminus \{x\}\} \\
&= \text{“since } n \notin deps_0 x \equiv x \notin sups_0 n\text{”} \\
&\quad \{n : nodes_0 \setminus (\{x\} \cup deps_0 x) \bullet n \mapsto sups_0 n\} \\
&\quad \cup \{n : deps_0 x \bullet n \mapsto sups_0 n \setminus \{x\}\} \\
&= \{n : nodes_0 \setminus \{x\} \bullet n \mapsto sups_0 n\} \oplus \\
&\quad \{n : deps_0 x \bullet n \mapsto sups_0 n \setminus \{x\}\} \\
&= (\{x\} \triangleleft sups_0) \oplus \{n : deps_0 x \bullet n \mapsto sups_0 n \setminus \{x\}\}
\end{aligned}$$

Thus the new concrete specification for *RemoveNode* is:

$$nodes, deps, sups : \left[\begin{array}{l} x \in nodes \\ sups\ x = \emptyset \end{array} , \begin{array}{l} nodes = nodes_0 \setminus \{x\} \\ deps = \{x\} \triangleleft deps_0 \\ sups = (\{x\} \triangleleft sups_0) \oplus \\ \{n : deps_0 x \bullet n \mapsto sups_0 n \setminus \{x\}\} \end{array} \right]$$

CanAdd Transforming the post-condition:

$$\begin{aligned}
b &\Leftrightarrow (y \neq x) \\
&\equiv b \Leftrightarrow (x \neq y \wedge (y, x) \notin (mk_rel\ deps)^+)
\end{aligned}$$

gives *CanAdd* as $b : \left[\{x, y\} \subseteq nodes , \begin{array}{l} b \Leftrightarrow \\ (x \neq y \wedge (y, x) \notin (mk_rel\ deps)^+) \end{array} \right]$

This can be refined³ procedurally:

$$\begin{aligned}
&\sqsubseteq \text{if } x = y \rightarrow b := \text{false} \\
&\quad \boxed{x \neq y \rightarrow b : \left[\{x, y\} \subseteq nodes , b \Leftrightarrow (y, x) \notin (mk_rel\ deps)^+ \right]} \triangleleft \\
&\quad \text{fi} \\
&= b : \left[\{x, y\} \subseteq nodes , b \Leftrightarrow y \notin (mk_rel\ sups)^+(\{x\}) \right]
\end{aligned}$$

³the \triangleleft symbol in the right margin is a refinement marker indicating that just this component of the program is refined in the next step.

$$\begin{aligned}
&\sqsubseteq \text{var } ds : \mathbb{F} X \bullet \\
&\quad ds : \left[\{x, y\} \subseteq nodes, y \in nodes \wedge ds = (mk_rel\ sups)^+(\{x\} \uparrow) \right]; \triangleleft \\
&\quad b : \left[y \in nodes \wedge ds = (mk_rel\ sups)^+(\{x\} \uparrow), b \Leftrightarrow y \notin ds \right] \\
&\sqsubseteq Dependents(x, ds)
\end{aligned}$$

The remaining specification statement requires a set membership test.

AddDependence Transforming the pre-condition gives:

$$\begin{aligned}
&\{x, y\} \subseteq nodes \wedge (y, x) \notin dir_dep_on^* \\
&\equiv \{x, y\} \subseteq nodes \wedge y \neq x \wedge (y, x) \notin (mk_rel\ deps)^+
\end{aligned}$$

The post-condition $dir_dep_on = dir_dep_on_0 \cup \{(x, y)\}$ is transformed by computing the required change in $deps$ and $sups$.

$$\begin{aligned}
deps &= \{n : nodes \bullet n \mapsto dir_dep_on(\{n\} \uparrow)\} \\
&= \{n : nodes \bullet n \mapsto (dir_dep_on_0 \cup \{(x, y)\})(\{n\} \uparrow)\} \\
&= \{n : nodes \bullet n \mapsto dir_dep_on_0(\{n\} \uparrow)\} \oplus \\
&\quad \{x \mapsto dir_dep_on_0(\{x\} \uparrow) \cup \{y\}\} \\
&= deps_0 \oplus \{x \mapsto dir_dep_on_0(\{x\} \uparrow) \cup \{y\}\} \\
&= deps_0 \oplus \{x \mapsto deps_0\ x \cup \{y\}\}
\end{aligned}$$

A symmetric argument for $sups$ leads to the new specification for *AddDependence*:

$$\begin{aligned}
&deps : \left[\begin{array}{l} \{x, y\} \subseteq nodes \\ x \neq y \end{array} \right. \\
&sup : \left. \left[(y, x) \notin (mk_rel\ deps)^+, \quad \begin{array}{l} deps = deps_0 \oplus \{x \mapsto deps_0\ x \cup \{y\}\} \\ sups = sups_0 \oplus \{y \mapsto sups_0\ y \cup \{x\}\} \end{array} \right] \right]
\end{aligned}$$

Supporters Transforming the post-condition gives:

$$\begin{aligned}
nds &= \{n : nodes \mid x \succ n\} \\
&\equiv nds = \{n : nodes \mid (x, n) \in (mk_rel\ deps)^+\} \\
&\equiv nds = (mk_rel\ deps)^+(\{x\} \uparrow)
\end{aligned}$$

Thus *Supporters* becomes $nds : \left[x \in nodes, nds = (mk_rel\ deps)^+(\{x\} \uparrow) \right]$.

It is convenient to proceed now with procedural refinement of this specification, defining a procedure *RecSup*. This procedure is invoked recursively to collect all supporters. The set $SS(x)$ is the supporter set of the node x .

$$\begin{aligned}
&\sqsubseteq SS(x) \hat{=} (mk_rel\ deps)^+(\{x\}) \bullet \\
&\quad \left\| \begin{array}{l} \mathbf{procedure}\ RecSup(\mathbf{value}\ x : X) \hat{=} \\ \quad \mathit{nds} : [\mathit{nds} = \mathit{nds}_0 \cup SS(x)] \bullet \quad (i) \\ \quad \mathit{nds} : [x \in \mathit{nodes} , \mathit{nds} = SS(x)] \quad (ii) \end{array} \right. \\
&\quad \left. \right\|
\end{aligned}$$

The body of the block can be refined using sequential composition to invoke *RecSup*.

$$\begin{aligned}
(ii) \quad &\sqsubseteq \mathit{nds} := \{ \}; \\
&\quad RecSup(x)
\end{aligned}$$

Refinement of the body of the procedure *RecSup* is guided by the well-known depth-first search strategy of graph traversal. The next few steps involve:

1. defining a name for the recursion block *RSup* and a logical constant $c1$,
2. introducing a local variable ds containing the set of direct supporters that have been processed,
3. introducing a logical constant $nds0$ representing the initial value of nds , and
4. using the sequential composition rule prior to introducing a loop.

$$\begin{aligned}
(i) \quad &\sqsubseteq \mathbf{re}\ RSup \hat{=} \{0 \leq \#SS(x) < c1\} \quad \mathit{nds} : [\mathit{nds} = \mathit{nds}_0 \cup SS(x)] \bullet \\
&\quad \mathit{nds} : [\#SS(x) = c1 , \mathit{nds} = \mathit{nds}_0 \cup SS(x)] \\
&\sqsubseteq \mathbf{var}\ ds : \mathbb{F} X; \\
&\quad \mathbf{con}\ nds0 : \mathbb{F} X \bullet \\
&\quad \mathit{nds}, ds : [\#SS(x) = c1 \wedge \mathit{nds} = \mathit{nds}_0 , \mathit{nds} = \mathit{nds}_0 \cup SS(x)]
\end{aligned}$$

$$\begin{aligned}
\sqsubseteq \quad & I \cong \#SS(x) = c1 \wedge nds = nds0 \cup (mk_rel\ deps)^*(| \ deps(x) \setminus ds |) \bullet \\
& ds := deps(x); \\
& nds, ds: [I, I \wedge ds = \{\}] \quad (iii)
\end{aligned}$$

The previous refinement step relies on the identity:

$$(mk_rel\ deps)^+(| \{x\} |) = (mk_rel\ deps)^*(| \ deps(x) |)$$

Refining (iii) to a loop with variant I and invariant $\#ds$ gives:

$$\begin{aligned}
(iii) \quad & \sqsubseteq \quad \mathbf{do} \ ds \neq \{\} \rightarrow \\
& \hspace{15em} nds, ds: [ds \neq \{\}, I, \#ds < \#ds_0] \quad \triangleleft \\
& \mathbf{od}
\end{aligned}$$

The loop body is refined by introducing a local variable k (representing an unprocessed, direct supporter of x) and applying the *following-assignment* rule.

$$\begin{aligned}
\sqsubseteq \quad & \mathbf{var} \ k : X \bullet \\
& nds, k: [ds \neq \{\} \wedge I, I_{[ds \setminus (ds \setminus \{k\})]} \wedge k \in ds]; \quad \triangleleft \\
& ds := ds \setminus \{k\}
\end{aligned}$$

The remaining specification is split into an assignment of any unprocessed direct supporter to k and a specification statement representing the updating of nds .

$$\begin{aligned}
\sqsubseteq \quad & k: [ds \neq \{\}, k \in ds]; \\
& nds: [k \in ds \wedge I, I_{[ds \setminus (ds \setminus \{k\})]}] \quad \triangleleft
\end{aligned}$$

Since the node k may have already have been processed via another path, an if command tests for this possibility. This case requires no further processing since all the supporters of k would be members of nds already.

$$\begin{aligned}
\sqsubseteq \quad & \mathbf{if} \ k \in nds \rightarrow \mathbf{skip} \\
& \quad \mathbf{[]} \ k \notin nds \rightarrow nds: [k \notin nds \wedge k \in ds \wedge I, I_{[ds \setminus (ds \setminus \{k\})]}] \quad \triangleleft \\
& \mathbf{fi}
\end{aligned}$$

At this stage, a test for circularities could be made but we have chosen not to do so since the pre-conditions exclude creating a graph with any circularities. As well, the test adds complexity to the development.

The guarded command is refined using the following assignment rule:

$$\sqsubseteq \text{nds} : \left[k \notin \text{nds} \wedge k \in ds \wedge I, I_{[\text{nds}, ds \setminus \text{nds} \cup \{k\}, (ds \setminus \{k\})]} \right]; \quad \triangleleft$$

$$\text{nds} := \text{nds} \cup \{k\}$$

Expanding and simplifying the last specification statement gives

$$\sqsubseteq \text{nds} : \left[\#SS(x) = c1, \text{nds} = \text{nds}_0 \cup SS(k) \right]$$

Since k is a supporter of x , $SS(k) \subset SS(x)$ and hence

$$\sqsubseteq \{0 \leq \#SS(k) < c1\} \quad \text{nds} : \left[\text{nds} = \text{nds}_0 \cup SS(k) \right]$$

$$\sqsubseteq [\text{value } x : X \setminus k] \bullet$$

$$\{0 \leq \#SS(k) < c1\} \quad \text{nds} : \left[\text{nds} = \text{nds}_0 \cup SS(x) \right]$$

$$= \text{RSup}$$

Collecting all the refinement steps for the *Supporters* operation gives the code in Figure 4.

SomeDirectDependent Transforming the pre-condition gives:

$$x \in \text{ran } \text{dir_dep_on}$$

$$\equiv x \in \text{ran} \{n1, n2 : \text{nodes} \mid n1 \in \text{sups } n2\}$$

$$\equiv (\exists n1 : \text{nodes} \bullet n1 \in \text{sups } x)$$

$$\equiv \text{sups } x \neq \emptyset$$

while transforming the post-condition gives:

$$(node, x) \in \text{dir_dep_on}$$

$$\equiv node \in \text{sups } x$$

thus *SomeDirectDependent* becomes $node : \left[\text{sups } x \neq \emptyset, node \in \text{sups } x \right]$.

```

procedure Supporters(value  $x : X$ ; result  $nds : \mathbb{F} X$ )  $\hat{=}$ 
  — Pre :  $x \in nodes$ 
  — Post :  $nds = (mk\_rel\ deps)^+(\{x\})$ 

[[ procedure RecSup(value  $x : X$ )  $\hat{=}$ 
  [[ var  $ds : \mathbb{F} X \bullet$ 
     $ds := deps(x)$ ;
    do  $ds \neq \{\}$   $\rightarrow$ 
      [[ var  $k : X \bullet$ 
         $k : [ds \neq \{\}, k \in ds]$ ;
        if  $k \in nds \rightarrow$  skip
        [[  $k \notin nds \rightarrow$  RecSup( $k$ );
           $nds := nds \cup \{k\}$ 
        fi;
         $ds := ds \setminus \{k\}$ 
      ]]
    od
  ]]
   $nds := \{\}$ ;
  RecSup( $x$ )
]]

```

Figure 4: Refinement result for the *Supporters* operation

5 New data representation — step 2

The second data refinement step replaces both the *nodes* set and the *deps* and *sup*s functions with a single function *nodemap* : $X \leftrightarrow (\mathbb{F} X \times \mathbb{F} X)$. This function associates with each node a pair of sets: the set of nodes that this node directly depends on and the set of nodes that this node directly supports. For the example in Figure 3, *nodemap* would have the value:

$$\left\{ \begin{array}{l} a \mapsto (\{b, c\}, \{\}), \\ b \mapsto (\{d\}, \{a\}), \\ c \mapsto (\{d, e\}, \{a\}), \\ d \mapsto (\{\}, \{b, c\}), \\ e \mapsto (\{\}, \{c\}) \end{array} \right\}$$

The abstraction invariant is again functional:

$$\begin{aligned} nodes &= \text{dom } nodemap \\ deps &= \{m : nodemap \bullet \pi_1 m \mapsto \pi_1(\pi_2 m)\} \\ sups &= \{m : nodemap \bullet \pi_1 m \mapsto \pi_2(\pi_2 m)\} \end{aligned}$$

Since the expressions for *nodes*, *deps* and *sup*s do not simplify easily, we continue to use these terms as abbreviations. Thus the concrete invariant remain unchanged (although the conjuncts $\text{dom } deps = nodes$ and $\text{dom } sups = nodes$ are unnecessary). We only show the refinement of some operations.

Performing the data refinement calculations on the *RemoveNode* operation gives the following specification.

$$\begin{aligned} &\mathbf{procedure} \text{ } RemoveNode(\mathbf{value} \ x : X) \hat{=} \\ &nodemap : \left[\begin{array}{l} x \in nodes \quad \quad \quad nodemap = \{x\} \triangleleft nodemap_0 \oplus \\ sups \ x = \emptyset \ , \ \{n : deps_0 \ x \bullet n \mapsto (deps_0 \ n, sups_0 \ n \setminus \{x\})\} \end{array} \right]; \end{aligned}$$

It is convenient to apply some procedural refinement by introducing a local variable and expanding the function override into an explicit loop.

$$\begin{aligned} \sqsubseteq \quad &nodemap : \left[\begin{array}{l} x \in nodes \quad \quad \quad nodemap = nodemap_0 \oplus \\ sups \ x = \emptyset \ , \ \{n : deps_0 \ x \bullet n \mapsto (deps_0 \ n, sups_0 \ n \setminus \{x\})\} \end{array} \right]; \triangleleft \\ &nodemap := \{x\} \triangleleft nodemap \end{aligned}$$

```


$$\sqsubseteq \text{var } ds : \mathbb{F} X \bullet$$


$$ds := \text{deps}(x);$$


$$ds, \text{nodemap} : \left[ \begin{array}{l} x \in \text{nodes} \\ \text{sup}s x = \emptyset \end{array} , \{n : \text{deps}_0 x \bullet n \mapsto (\text{deps}_0 n, \text{sup}s_0 n \setminus \{x\})\} \right] \triangleleft$$


$$\sqsubseteq \text{con } \text{olddeps} : x \mapsto \mathbb{F} X \bullet$$


$$ds, \text{nodemap} : \left[ \begin{array}{l} x \in \text{nodes} \\ \text{sup}s x = \emptyset \\ \text{olddeps} = \text{deps} \end{array} , \{n : \text{deps}_0 x \bullet n \mapsto (\text{deps}_0 n, \text{sup}s_0 n \setminus \{x\})\} \right]$$


$$\sqsubseteq \text{invariant } \forall n : \text{olddeps } x \setminus ds \bullet x \notin \text{sup}s n \quad \text{variant } \#ds$$


$$\text{do } ds \neq \emptyset \rightarrow$$


$$\quad \left[ \begin{array}{l} \text{var } k : X \bullet \\ k : [ds \neq \emptyset, k \in ds]; \\ \text{nodemap} : \left[ \begin{array}{l} \text{nodemap} = \text{nodemap}_0 \oplus \\ \{k \mapsto (\text{deps}_0 k, \text{sup}s_0 k \setminus \{x\})\} \end{array} \right]; \\ ds := ds \setminus \{k\} \end{array} \right]$$


$$\quad \left. \right]$$


$$\text{od}$$


```

It is also convenient to refine the *AddDependence* procedure:

```

procedure AddDependence(value  $x, y : X$ )  $\hat{=}$ 

$$\text{nodemap} : \left[ \begin{array}{l} \{x, y\} \subseteq \text{nodes} \\ x \neq y \\ (y, x) \notin (\text{mk\_rel } \text{deps})^+ \end{array} , \begin{array}{l} \text{nodemap} = \text{nodemap}_0 \\ \oplus \{x \mapsto (\text{deps}_0 x \cup \{y\}, \text{sup}s x)\} \\ \oplus \{y \mapsto (\text{deps } y, \text{sup}s_0 y \cup \{x\})\} \end{array} \right]$$


```

by performing the two updates to the *nodemap* as sequential steps (recognising that this is unlikely to be implemented as a single monolithic operation). Of course the order of the two steps is completely arbitrary.

```


$$\sqsubseteq \text{nodemap} : \left[ \text{nodemap} = \text{nodemap}_0 \oplus \{x \mapsto (\text{deps}_0 x \cup \{y\}, \text{sup}s x)\} \right];$$


$$\text{nodemap} : \left[ \text{nodemap} = \text{nodemap}_0 \oplus \{y \mapsto (\text{deps } y, \text{sup}s_0 y \cup \{x\})\} \right]$$


```

Summarising the current state of the module after the second data refinement and some subsequent procedural refinement:

module *DepManSys2* $\hat{=}$

var *nodemap* : $X \leftrightarrow (\mathbb{F} X \times \mathbb{F} X)$;

abbreviations

nodes = $\text{dom } \textit{nodemap}$

deps = $\{m : \textit{nodemap} \bullet \pi_1 m \mapsto \pi_1(\pi_2 m)\}$

*sup*s = $\{m : \textit{nodemap} \bullet \pi_1 m \mapsto \pi_2(\pi_2 m)\}$

invariant

$(\text{dom } \textit{deps}) = \textit{nodes} \wedge (\text{dom } \textit{sup}s) = \textit{nodes}$

$\wedge \bigcup \text{ran } \textit{deps} \subseteq \textit{nodes} \wedge \bigcup \text{ran } \textit{sup}s \subseteq \textit{nodes}$

$\wedge \textit{mk_rel } \textit{deps} = (\textit{mk_rel } \textit{sup}s)^\sim \wedge (\textit{mk_rel } \textit{deps})^+ \cap \text{id } X = \emptyset$

initially *nodemap* = \emptyset

procedure *RemoveNode*(**value** $x : X$) $\hat{=}$

— *Pre* : $x \in \textit{nodes} \wedge \textit{sup}s \ x = \emptyset$

— *Post* : $\textit{nodemap} = \{x\} \triangleleft \textit{nodemap}_0 \oplus$

$\{n : \textit{deps}_0 \ x \bullet n \mapsto (\textit{deps}_0 \ n, \textit{sup}s_0 \ n \setminus \{x\})\}$

\llbracket **var** $ds : \mathbb{F} X \bullet$

$ds := \textit{deps}(x)$;

do $ds \neq \emptyset \rightarrow$

\llbracket **var** $k : X \bullet$

$k : [ds \neq \emptyset, k \in ds]$;

$\textit{nodemap} : \left[\begin{array}{l} \textit{nodemap} = \textit{nodemap}_0 \oplus \\ \{k \mapsto \textit{deps}_0 \ k, \textit{sup}s_0 \ k \setminus \{x\}\} \end{array} \right]$;

$ds := ds \setminus \{k\}$

\rrbracket

od

\rrbracket ;

$\textit{nodemap} := \{x\} \triangleleft \textit{nodemap}$

```

procedure AddDependence(value  $x, y : X$ )  $\hat{=}$ 
  — Pre :  $\{x, y\} \subseteq \text{nodes} \wedge x \neq y \wedge (y, x) \notin (\text{mk\_rel } \text{deps})^+$ 
  — Post :  $\text{nodemap} = \text{nodemap}_0 \oplus \{x \mapsto (\text{deps}_0 x \cup \{y\}, \text{sup}_s x)\}$ 
     $\oplus \{y \mapsto (\text{deps } y, \text{sup}_s y \cup \{x\})\}$ 

  nodemap:  $\left[ \text{nodemap} = \text{nodemap}_0 \oplus \{x \mapsto (\text{deps}_0 x \cup \{y\}, \text{sup}_s x)\} \right];$ 
  nodemap:  $\left[ \text{nodemap} = \text{nodemap}_0 \oplus \{y \mapsto (\text{deps } x, \text{sup}_s y \cup \{x\})\} \right];$ 

```

```

procedure Supporters(value  $x : X$ ; result  $\text{nds} : \mathbb{F} X$ )  $\hat{=}$ 
  — Pre :  $x \in \text{nodes}$ 
  — Post :  $\text{nds} = (\text{mk\_rel } \text{deps})^+(\{x\})$ 

```

```

 $\llbracket$ 
  procedure RecSup(value  $x : X$ )  $\hat{=}$ 
     $\llbracket$  var  $\text{ds} : \mathbb{F} X \bullet$ 
       $\text{ds} := \text{deps}(x);$ 
      do  $\text{ds} \neq \{\}$   $\rightarrow$ 
         $\llbracket$  var  $k : X \bullet$ 
           $k : \left[ \text{ds} \neq \{\}, k \in \text{ds} \right];$ 
          if  $k \in \text{nds} \rightarrow$  skip
           $\llbracket$   $k \notin \text{nds} \rightarrow$  RecSup( $k$ );
             $\text{nds} := \text{nds} \cup \{k\}$ 
          fi;
           $\text{ds} := \text{ds} \setminus \{k\}$ 
         $\rrbracket$ 
      od
     $\rrbracket$ 
     $\text{nds} := \{\};$ 
    RecSup( $x$ )
   $\rrbracket;$ 

```

```

procedure CanAdd(value  $x, y : X$ ; result  $b : \mathbb{B}$ )  $\hat{=}$ 
  — Pre :  $\{x, y\} \subseteq nodes$ 
  — Post :  $b \Leftrightarrow (x \neq y \wedge (y, x) \notin (mk\_rel\ deps)^+)$ 

  if  $x = y \rightarrow b := \text{false}$ 
   $\llbracket$   $x \neq y \rightarrow$ 
     $\llbracket$  var  $ds : \mathbb{F} X \bullet$ 
      Dependents( $x, ds$ );
       $b : [y \in nodes, b \Leftrightarrow y \notin ds]$ 
     $\rrbracket$ 
   $\rrbracket$ 
  fi;

procedure SomeDirectDependent(value  $x : X$ ; result  $node : X$ )  $\hat{=}$ 
   $node : [sups\ x \neq \emptyset, node \in sups\ x]$ 
end

```

6 Conclusions

This report demonstrates how the refinement calculus can be used for both procedural and data refinement. The results of this refinement have been used to develop an Eiffel implementation.

References

- [1] R.-J. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.
- [2] W. Chen and J.T. Udding. Towards a calculus of data refinement. In J.L.A. van de Snepscheut, editor, *Mathematics for Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 197–218, 1989.
- [3] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

- [4] Paul Gardiner and Carroll Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143–162, 1991.
- [5] S. King. Z and the refinement calculus. In *VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*, pages 164–188. Springer-Verlag, 1990.
- [6] Peter A. Lindsay. The dependency management system case study. Technical Report 94-01, Software Verification Research Centre, The University of Queensland, December 1994.
- [7] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.
- [8] C. Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.
- [9] J. M. Morris. Laws of data refinement. *Acta Informatica*, 26, 1989.
- [10] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *The Science of Computer Programming*, 9(3):287–306, December 1987.