Enhancing the template mechanism

Laurence Tratt, Tony Clark
laurie@tratt.net, anclark@dcs.kcl.ac.uk

King's College London

Abstract. MDA needs to utilize a variety of modelling languages in order to be successful. It is widely acknowledged that there is a need to be able to systematically create such modelling languages with much of the research in this area being in the context of transforming the UML into a 'family of languages'. Although some progress has been made towards this goal with the introduction of technologies such as templates and package extension, the overall approach has been rather simplistic and coarse-grained. In this paper we introduce, and formally define via a specialized calculus, an extension to the template mechanism which allows more complex templates to be constructed.

1 Introduction

The OMG's Model Driven Architecture (MDA) [SO00] initiative is aimed at addressing the real-life difficulties of developing large systems by utilizing modelling in various ways. One of the keys to MDA's success will be its ability to utilize differing modelling languages. In order to achieve this goal there needs to be a systematic way of constructing modelling languages and whilst many of the modelling languages will probably be based on the de facto modelling standard UML [OMG01], the mechanism needs to be flexible enough to construct modelling languages of any type.

The beginnings of such work have been done in the UML field as there has been an awareness for some time of the need to turn the UML – as of version 1.4, something of a monolith – into a 'family of languages'. One particular method for achieving this aim and that has its roots in work such as [DW98,CEK+00,CEK02a] is the basis for a submission for UML 2.0 [CEK02b] (hereon referred to as the '2U submission'). At the heart of this method is the notion of using templates to specify certain parts of modelling languages and we take this application of templates as the basis for our work. Even in their current rather limited form, templates have proved themselves to be a useful and practical mechanism for constructing modelling languages. In the subsequent section we show where we have identified areas where the current template mechanism is not as powerful as will be needed in the near future.

Our work has aimed at enhancing the template mechanism with new facilities. In order to do this, we have developed a fairly low-level calculus for templates which allows us to precisely define our new template mechanism. We feel that a formal definition is critical to the long-term usefulness of such a mechanism. We

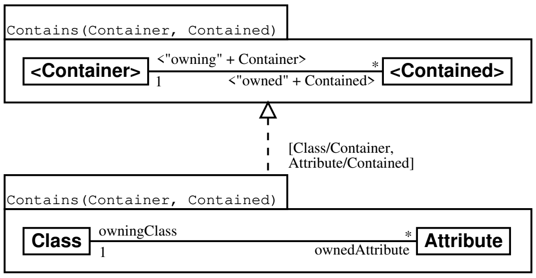

Fig. 1. Simple template instantiation

show a number of examples, including a simple demonstration that our formalism encapsulates current template technology, which demonstrate some of the new facilities the calculus provides.

2 Templates

2.1 Current template technology

Templates were popularized in [DW98] and are intended to capture a specific pattern of usage: they are effectively a macro mechanism for modelling. Templates are currently implemented as parameterized packages; in order to use a template, a child package instantiates[1] the template package via a special variant of package extension, giving it certain arguments which are substituted in the templates contents upon instantiation. Figure 1 shows a particularly simple template, which captures the notion of one modelling element 'containing' another, being instantiated to produce a model where classes contain attributes. Note that we have shown the *result* of the instantiation in the child package.

[CEK02b] contains a number of templates, most far more complex than that of figure 1, intended to be used together to create modelling languages. Unsurprisingly, given that documents intention to propose a new version of the UML, many of the templates contained therein are specifically targeted at creating UML-like modelling languages. Such templates are unlikely to be of much use in creating, for example, a more minimalistic modelling language accommodating as they do the bulk gained by the UML over the years. Furthermore, it seems very unlikely that they will be of any use at all in creating a modelling language intended to model a problem domain other than the UML. The reason for the templates' lack of applicability outside of their explicitly stated domain is largely because there is no flexibility in the template mechanism. If a template doesn't precisely fit ones specific purpose, one has no choice but to create by hand the chunk of modelling language that was so nearly created by the template. The reality is that modelling languages often need subtle variations on any given theme (for an example see [TCE02] which enumerates a number of different styles of

[1] The nomenclature for this term is currently somewhat fluid. 'Extends', 'stamping out' and even 'generalizes' are terms in current use to varying degrees.

subclassing found in programming languages and then presents differing templates to construct the relevant chunks of modelling languages in a fine-grained manner).

At a high level, one can imagine two complementary schemes for resolving this lack of flexibility. The first is to construct libraries of related templates (as in [TCE02]) and then present a control mechanism for their applicability (perhaps in a similar way as transformation units control the application of graph transformations [AEH$^+$99]). The second is by extending the template mechanism itself e.g. by building control structures into templates, allowing any given template to vary its output depending on its input arguments. Each scheme raises certain issues and presents its own challenges and neither has been investigated before. In this paper, we focus on the latter of these two schemes: enhancing the template mechanism.

2.2 An enhanced template mechanism

Having argued that there is a need for a more flexible template mechanism, the question arises as to exactly what such a mechanism needs to incorporate. Because we can really only consider ourselves to be at the beginning of the 'template age', a chicken-and-egg paradox is evident. The lack of flexibility in templates has meant that hitherto little thought has been given to precisely what facilities would be desirable. We therefore restrict our discussion to a few very simple enhancements which we believe have universal applicability and await further debate as to which more esoteric enhancements may or may not be desirable.

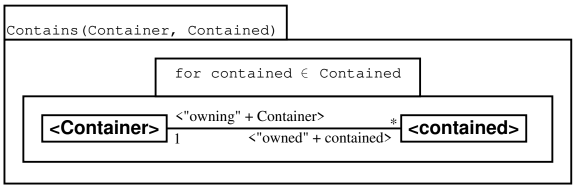

Fig. 2. A template including a *for* loop

One of the most interesting enhancement we have added is that of a *for* loop. The intention is to allow a part of a template to generate model chunks based on iterating over a set. Figure 2 presents a simple but compelling example based on a simple variation on the standard container template of figure 1. The slightly modified UML syntax uses a box with a centred title bar (unlike a package whose title bar is positioned to the left of the main box) to denote the extra syntax. The idea is that a Container often contains more than one element: it therefore seems rather unnatural[2] to instantiate the template more than once

[2] Although we do not follow this up in this paper, the fact that the template is constructed in one, rather than many, steps opens up the possibility of combining the

to capture this rather fundamental notion. In this example, one passes a single Container and a set of Contained elements (this is not made explicit in the diagram as template arguments are currently untyped); the template then sets up the standard notion of containment between the single Container and set of Contained elements via the for loop. Figure 3 shows an example of instantiating this template.

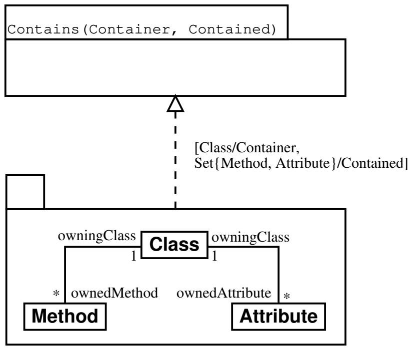

Fig. 3. Instantiation of the enhanced Container template

Perhaps the simplest enhancements we have made is that of a conditional statement. A contrived example is presented in figure 4. In this example, depending on whether the boolean *addC* is true or false, A will be associated with C or D respectively. Our syntax for a conditional is much as with the for loop of figure 2 utilizing a box with a centred title bar; in the conditional, the top half of the box is the 'then' clause and the bottom half the 'else' clause.

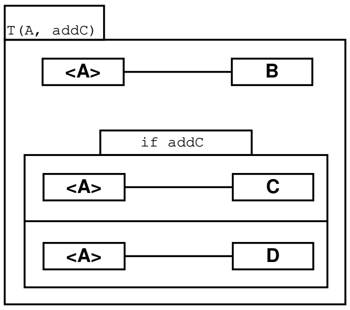

Fig. 4. A template utilizing a conditional

templates arguments together e.g. in a constraint in ways that would not otherwise be possible.

3 A calculus for templates

In this section we define a calculus for templates that allows one to precisely specify a template. We present a simple encoding of standard template technologies into our calculus as well as present new constructs which enhance the template concept into new areas. The calculus itself is small and side-effect free, allowing us to relatively simply define its syntax and semantics and encode more powerful constructs in terms of the primitives. It can be intuitively viewed as an extension of the λ calculus with records and an operator to merge records together.

The calculus and its definition are inspired by, and largely follow the same patterns as, the **imp**ς [AC96] and **par**ς [Han99] calculi. The interested reader is referred to both publications, particularly the former, for more detail on such definitions than we have space for.

3.1 Syntax

We define the syntax of a small calculus for templates in the following table (in later sections we allow s followed by any number of the $'$ character to range over strings).

Syntax of the calculus for templates

$t,\ u,\ v$ ::=	templates
$\Lambda(n_i\ ^{i\in 1\ldots a}).t$	template definition
$t\ u_i\ ^{i\in 1\ldots a}$	application
$let\ x_i\ =\ t_i\ ^{i\in 1\ldots a}\ in\ u$	let
$letrec\ x_i\ =\ t_i\ ^{i\in 1\ldots a}\ in\ u$	recursive let
$if\ t\ then\ u\ else\ v$	if
$[t_i\ =\ u_i\ ^{i\in 1\ldots a}]$	record
n	variable
$Set\{t_i\ ^{i\in 1\ldots a}\}$	set
k	constant
k ::=	constant
s	string
b	bool

There are a number of predefined primitive functions: $\oplus$, $=$, $+$, $\cup$, *select* and *remove*. We use these functions unsugared in prefix notation.

Scoping rules In a template definition $\Lambda(n_i\ ^{i\in 1\ldots a}).t$, $n_i\ ^{i\in 1\ldots a}$ are bound and their scope is t. In a let $let\ x_i\ =\ t_i\ ^{i\in 1\ldots a}\ in\ u$, $x_i\ ^{i\in 1\ldots a}$ are bound and their scope is u. In a recursive let $letrec\ x_i\ =\ t_i\ ^{i\in 1\ldots a}\ in\ u$, $x_i\ ^{i\in 1\ldots a}$ are bound and their scope is u. We denote the free variables of a term a by $fv(a)$. A term t is said to be closed iff $fv(t)\ =\ \emptyset$. We denote a capture-avoiding substitution of u for each free occurrence of n in u as $u\{\!\{n \leftarrow u\}\!\}$.

3.2 Semantics

Informal semantics The informal semantics of many of the calculus' terms are self-explanatory; for the slightly less obvious terms, the following notes should give one an intuitive feel:

- A template definition $\Lambda(n_i\ ^{i\in 1\ldots n}).t$ is effectively a function definition.
- An application $t(u_i\ ^{i\in 1\ldots n})$ applies a number of arguments to a template definition.
- An application $n(u_i\ ^{i\in 1\ldots n})$ applies a number of arguments to a named built in function.
- A let $let\ x_i\ =\ t_i\ ^{i\in 1\ldots a}\ in\ u$ evaluates in parallel its arguments and then substitutes the results into u.
- A recursive let $letrec\ x_i\ =\ t_i\ ^{i\in 1\ldots a}\ in\ u$ evaluates its arguments in parallel allowing arguments to refer to themselves or other arguments, and then substitutes the results into u.
- A record $[t_i\ =\ u_i\ ^{i\in 1\ldots a}]$ is an element which contains named elements. We use this to abstract away the difference between e.g. classes and packages.

The calculus has the following built in functions:

- A merge $\oplus\ t\ u$ takes two records – where the named elements in both records must have reduced to a string – and merges u and t into a new record. The merge is transitive and reflexive.
- A comparison $=\ t\ u$ compares two terms of the same type (currently strings, sets or records) and returns a boolean value.
- An addition $+\ t\ u$ adds two strings of the same type together and returns the result.
- A union $\cup\ t\ u$ unions two sets together and returns the result.
- A select $select\ t$ picks at random an element from the set t.
- A removal $remove\ t\ u$ removes the element u from the set t.

Operational semantics As in [AC96,Han99] we present an operational semantics for our calculus via a reduction system. The reduction system reduces every closed term to a result (sometimes called normal form). In our definition we assume the existence of primitive type functions such as *isString(s)* to check the type of a given term, and a special function *isResult(t)* which returns true if t is a result.

Our calculus has the following results:

Results

r ::=		results
$\Lambda(n_i\ ^{i\in 1\ldots a}).t$		template definition
$[t_i\ =\ u_i\ ^{i\in 1\ldots a}]$	$isResult(t_i)\quad \forall i \in 1\ldots a$	record
n		variable
$Set\{t_i\ ^{i\in 1\ldots a}\}$		set
k		constant

The reduction relation is denoted $\rightarrow$. The following table defines the reduction rules (with the exception of the $\oplus$ operator which is defined in the following section):

Reduction rules

(Red Result)

$$\frac{}{r \;\rightarrow\; r}$$

(Red Application)

$$\frac{}{\Lambda(n_i).t\; u_i\; ^{i\in 1\ldots a} \rightarrow t\{\!\{n_i \;\leftarrow\; u_i\; ^{i\in 1\ldots a}\}\!\}}$$

(Red Record 1)

$$\frac{\begin{array}{c} t_i \;\rightarrow\; v_i \quad \forall i \in 1\ldots a \\ isString(v_i) \;\wedge\; isString(v_j) \;\wedge\; v_i \;\equiv\; v_j \quad i,j \in 1\ldots a \quad i \neq j \\ u_i \;\rightarrow\; w \\ u_j \;\rightarrow\; x \\ isRecord(w) \;\wedge\; isRecord(x) \end{array}}{[t_k \;=\; u_k\; ^{k\in 1\ldots a}] \;\rightarrow\; [v_k \;=\; u_k\; ^{k\in (1\ldots a)-\{i\}},\; v_i \;=\; \oplus\; w\; x]}$$

(Red Record 2)

$$\frac{\begin{array}{c} t_i \;\rightarrow\; v_i \quad \forall i \in 1\ldots a \\ isString(v_i) \;\wedge\; isString(v_j) \;\wedge\; v_i \;\equiv\; v_j \quad i,j \in 1\ldots a \quad i \neq j \\ u_i \;\rightarrow\; w \\ u_j \;\rightarrow\; x \\ \neg isRecord(w) \\ =\; w\; x \;\rightarrow\; True \end{array}}{[t_k \;=\; u_k\; ^{k\in 1\ldots a}] \;\rightarrow\; [v_k \;=\; u_k\; ^{k\in (1\ldots a)-\{i\}},\; v_i \;=\; w]}$$

(Red Record 3)

$$\frac{\begin{array}{c} t_i \;\rightarrow\; v_i \quad \forall i \in 1\ldots a \\ (=\; v_i\; v_j \;\rightarrow\; True) \;\Rightarrow\; \neg isString(v_i) \quad \forall i,j \in 1\ldots a \quad i \neq j \end{array}}{[t_i \;=\; u_i\; ^{i\in 1\ldots a}] \;\rightarrow\; [v_i \;=\; u_i\; ^{i\in 1\ldots a}]}$$

(Red Let)

$$\frac{t_i \;\rightarrow\; v_i \quad \forall i \in 1\ldots a}{let\; x_i \;=\; t_i\; ^{i\in 1\ldots a}\; in\; u \;\rightarrow\; u\{\!\{x_i \;\leftarrow\; v_i\; ^{i\in 1\ldots a}\}\!\}}$$

(Red LetRec)

$$\frac{t_i\{\!\{x_j \;\leftarrow\; v_j\}\!\} \;\rightarrow\; v_i \quad \forall i,j \in 1\ldots a}{letrec\; x_i \;=\; t_i\; ^{i\in 1\ldots a}\; in\; u \;\rightarrow\; u\{\!\{x_i \;\leftarrow\; v_i\; ^{i\in 1\ldots a}\}\!\}}$$

(Red If 1)

$$\frac{t \;\rightarrow\; True}{if\; t\; then\; u\; else\; v \;\rightarrow\; u}$$

(Red If 2)

$$\frac{t \;\rightarrow\; False}{if\; t\; then\; u\; else\; v \;\rightarrow\; v}$$

(Red String Plus)

$$\frac{\begin{array}{c} u \ \rightarrow \ s' \\ isString(s') \\ v \ \rightarrow \ s'' \\ isString(s'') \\ s''' \ = \ s' \ + \ s'' \end{array}}{+ \ u \ v \ \rightarrow \ s'''}$$

(Red String Equality)

$$\frac{\begin{array}{c} u \ \rightarrow \ s' \\ isString(s') \\ v \ \rightarrow \ s'' \\ isString(s'') \\ s''' \ = \ (s' \ \equiv \ s'') \end{array}}{= \ u \ v \ \rightarrow \ s'''}$$

(Red Set Union)

$$\frac{\begin{array}{c} u \ \rightarrow \ Set\{t_i \ ^{i \in 1 \ldots a}\} \\ v \ \rightarrow \ Set\{u_j \ ^{j \in 1 \ldots b}\} \\ w \ = \ Set\{t_i \ ^{i \in 1 \ldots a}\} \ \cup \ Set\{u_j \ ^{j \in 1 \ldots b}\} \end{array}}{\cup \ u \ v \ \rightarrow \ w}$$

(Red Select)

$$\frac{\begin{array}{c} u \ \rightarrow \ e \\ e \ = Set\{t_i \ ^{i \in 1 \ldots a}\} \\ w \ = \ t_i \quad i \in 1 \ldots a \end{array}}{select \ u \ \rightarrow \ w}$$

(Red Remove)

$$\frac{\begin{array}{c} u \ \rightarrow \ Set\{t_i \ ^{i \in 1 \ldots a}\} \\ v \ \rightarrow \ t_j \quad j \ \in \ 1 \ldots a \\ x \ = \ Set\{t_i \ ^{i \in (1 \ldots a) - \{j\}}\} \end{array}}{remove \ u \ v \ \rightarrow \ x}$$

(Red Set Equality)

$$\frac{\begin{array}{c} u \ \rightarrow \ Set\{t_i \ ^{i \in 1 \ldots a}\} \\ v \ \rightarrow \ Set\{u_j \ ^{j \in 1 \ldots b}\} \\ w \ = \ (Set\{t_i \ ^{i \in 1 \ldots a}\} \ \equiv \ Set\{u_j \ ^{j \in 1 \ldots b}\}) \end{array}}{= \ u \ v \ \rightarrow \ w}$$

(Red Record Equality)

$$\frac{\begin{array}{c} u \ \rightarrow \ [t_i \ = \ u_i \ ^{i \in 1 \ldots a}] \\ v \ \rightarrow \ [v_i \ = \ w_i \ ^{i \in 1 \ldots b}] \\ a = b \\ = \ t_i \ v_i \ \rightarrow \ True \quad \forall i \in 1 \ldots a \\ = \ u_i \ w_i \ \rightarrow \ True \quad \forall i \in 1 \ldots a \end{array}}{= \ u \ v \ \rightarrow \ True}$$

With the exception of the (Red Record) rules, the reduction rules are largely standard and undeserving of further examination. The (Red Record) rules reduce a record to a result. (Red Record 1) and (Red Record 2) merge labels of the same name together: if the values of the labels are both records, (Red Record 1) merges those two values together; otherwise the values of the labels must be the same (Red Record 2). (Red Record 3) merges two disparate objects together.

Definition of the $\oplus$ operator The $\oplus$ operator is rather harder to define than any other part of the calculus due to the complexities of properly coping with recursive structures (which are created by *letrec*). Consider the two packages of figure 5 and then imagine what the result of a package R which extends packages Q and R should be. It is clearly crucial that in the resulting package R, classes B and C point to the same class A – a naïve implementation would quite possibly create two A classes.

Therefore the situation we find ourselves in is that of figure 6 where packages P and R are represented by the top left and top right graphs respectively. The graphs explicitly show the recursive nature of the original package definitions. The bottom graph in the figure is what we wish the result of the $\oplus$ operator to

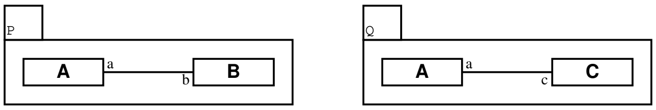

Fig. 5. Example packages

return, whereby there is only one node representing the class A to which both B and C point to. As we can easily represent records by graphs, it therefore seems natural to formalize this definition using graph transformations [AEH+99]. Unfortunately, a formal definition of this sort is more bulky than we have room for in this paper and we therefore leave the intuitive understanding of the $\oplus$ operator from figure 6 as the current definition.

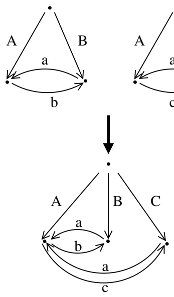

Fig. 6. Example packages represented as graphs

3.3 The calculus in action

Encoding attributes As our calculus has no direct notion of references we can not easily represent associations so instead we introduce a simple encoding for attributes. The encoding effectively embeds the notion of a 'type' into the value of an attribute and also stores whether the attribute is a 'plain' type or a 'set' type.

This encoding assumes we have a standard encoding of tuples (which can be trivially lifted from any text on the λ calculus) and functions to access the *nth* member of a tuple as well as two distinguishable constants *TYPE* and *SET-TYPE*.

Encoding attributes

Type(class)	$\triangleq$ (class, TYPE)
isType(pair)	$\triangleq$ = second(pair) TYPE
SetType(class)	$\triangleq$ (class, SETTYPE)
isSetType(pair)	$\triangleq$ = second(pair) SETTYPE

The Container template As a simple example, the standard Container template expressed in our calculus looks thus:

```
Λ(Container Contained).
  letrec
    c1 = ["owned" + Contained = SetType(c2)]
    c2 = ["owning" + Container = Type(c1) ]
  in
    [Container = c1,
    Contained = c2]
  end
```

Notice the use of the recursive let in order to generate the types correctly.

3.4 Type system

The syntax of the calculus for templates is rather general and allows a large number of terms to be written which are not well-formed. The obvious solution to this problem is to introduce a type system which restricts valid terms to a subset of the original. Unlike some calculi such as [Han99] whose type systems are able to very heavily constrain calculus terms – in that particular instance, being able to offer the guarantee that there will be no run time errors in a calculus term – our type system can make no such guarantees in the general case.

The reason for this is simple: records do not in general have a 'fixed detailed type' in our calculus since it is possible to rename components within such elements. This is unlike [AC96] where an object has a fixed number of immutable labels and can thus be assigned a type directly related to its constituent parts. As $\oplus$ operates on the contents of such elements it is not possible in general to determine whether a $\oplus\ t\ u$ will execute correctly at runtime since the precise contents of t and u may only be finally determined just before its execution. It is possible that more advanced analysis of the flow within a calculus term would allow one to determine at any given point the type of a record; this work is beyond our current scope.

Our type system, as with λ calculus type systems, assumes that template definition arguments are explicitly typed.

The types in our system are the following:

Types

$A,\ B$ ::=
- $Boolean$
- $String$
- $Record$
- $Set(A)$
- $A_i\ ^{i \in 1 \ldots a} \ \rightarrow\ B$ Template/Function

We are unable to give a more detailed type of *Record* for the reasons mentioned previously.

Our type system is based on the following three standard judgments:

Judgments

$E \vdash \diamond$	environment E is well-formed
$E \vdash A$	given E, type A is well-formed
$E \vdash a : A$	term a has type A in E

Our typing rules consist of a number of premises followed by a single judgment of the form $E \vdash a$ where E is an environment and a is an assertion (we follow an identical pattern to [AC96] to which the interested reader is directed for a more thorough explanation of the rules). In the interests of brevity rather than manually typing each of the calculus' built in functions, the environment E is assumed to contain the following types automatically: $\oplus$: *Record Record* $\rightarrow$ *Record*, $+$: *String String* $\rightarrow$ *String*, $=$: *String String* $\rightarrow$ *String*, $=$: *Set Set* $\rightarrow$ *String*, $\cup$: *Set Set* $\rightarrow$ *Set*, *select* $Set(A) \rightarrow A$ and *remove* $Set(A)\ B \rightarrow Set(A)$.

Noting that as in the typed λ calculus, template arguments must be explicitly typed, the typing rules are as follows:

Typing rules

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env n)
$$\frac{E \vdash A \quad n \notin dom(E)}{E,\ n : A \vdash \diamond}$$

(Type Boolean)
$$\frac{E \vdash \diamond}{E \vdash Boolean}$$

(Type String)
$$\frac{E \vdash \diamond}{E \vdash String}$$

(Type Template)
$$\frac{E \vdash A_i \quad \forall i \in 1\ldots a \qquad E \vdash B}{E \vdash A_i\ {}^{i\in 1\ldots a} \rightarrow B}$$

(Type Set)
$$\frac{E \vdash A}{E \vdash Set(A)}$$

(Type Record)
$$\frac{E \vdash \diamond}{E \vdash Record}$$

(Val n)
$$\frac{E',\ n : A,\ E'' \vdash \diamond}{E',\ n : A,\ E'' \vdash n : A}$$

(Val Template)
$$\frac{E,\ n_i : A_i\ {}^{i\in 1\ldots a} \vdash t : T}{E \vdash \Lambda(n_i : A_i\ {}^{i\in 1\ldots a}).t : A_i\ {}^{i\in 1\ldots a} \rightarrow T}$$

(Val Template Application)
$$\frac{E \vdash t : A_i\ {}^{i\in 1\ldots a} \rightarrow T \qquad E \vdash u_i : A_i \quad \forall i \in 1\ldots a}{E \vdash t\ u_i\ {}^{i\in 1\ldots a} : T}$$

(Val If)
$$\frac{E \vdash t : Boolean \qquad E \vdash u : T \qquad E \vdash v : T}{E \vdash if\ t\ then\ u\ else\ v : T}$$

(Val Let)
$$\frac{E \vdash t_i : A_i \quad \forall i \in 1\ldots n \qquad E,\ x_i : A_i\ {}^{i\in 1\ldots a} \vdash u : T}{let\ x_i = t_i\ {}^{i\in 1\ldots a}\ in\ u : T}$$

(Val LetRec)
$$\frac{E \vdash t_i : A_i \quad \forall i \in 1\ldots n \qquad E,\ x_i : A_i\ {}^{i\in 1\ldots a} \vdash u : T}{letrec\ x_i = t_i\ {}^{i\in 1\ldots a}\ in\ u : T}$$

Most of the typing rules are fairly simple. (Val Template) ensures that the parameters of a template definition are unique by taking advantage of the fact that (Env n) stipulates that an environment can not contain two terms of the same name.

As simple examples of our tying rules, the standard Container template of section 3.3 receives a type of *String Set*{*String*} $\rightarrow$ *Record* (assuming that the two arguments to the template are typed *Set* and *Set*{*String*} respectively) whilst $if\ = {'s1'}\ {'s2'}\ then\ {'s'}\ else\ True$ does not receive a type.

3.5 Enhancing the calculus with for loops

One facility that is notably lacking in current template technology is the ability to iterate over a set of items on input. We thus present an encoding of the calculus for templates enhanced with 'for' loops. Our for loops are slightly different than found in many programming languages, since the side effect free nature of the calculus would render a standard for loop useless. Each iteration of the for loop is expected to return a record and the records returned are merged together by the $\oplus$ operator.

Syntax of the calculus for templates enhanced with for loops

$t,\ u,\ v$ ::=	templates
...	as before
for n in t do u end	for

Our encoding is as follows:

```
⟦for x in L do B end⟧ ≜
  letrec
    t = ΛS.
      if = S ∅ then
        []
      else
        let
          x = select S
        in
          let
            S = remove S x
          in
            ⟦B⟧ ⊕ t(S)
          end
        end
      end
  in
    t(⟦L⟧)
```

```
19      end
20      for t ∉ fv(B)  S ∉ fv(B)
```

This encoding is essentially a recursive template which, on each iteration, randomly selects an element from the list L (line 9), removes that element from the list[3] (line 11) and merges it with the result of calling itself with the new list (line 13). When the list is empty, the recursion stops by returning an empty record (line 5) which also allows the $\oplus$ operator to act sensibly.

This allows us to update the Container template to take a set of contained elements:

```
Λ(Container Contained).
  for contained in Contained do
    letrec
      c1 = ["owned" + Contained = SetType(c2)]
      c2 = ["owning" + Container = Type(c1) ]
    in
      [Container = c1,
      Contained = c2]
    end
  end
```

It is worth noting how this updated template makes use of the *for* loop property that each iteration of the loop must return a record. In this instance, one can view each iteration as creating a new *contained* element and updating the *Contained* element.

3.6 Simple encoding of the 2U templates

In this section we present a simple encoding which demonstrates how the template calculus can be used to model the current 2U template mechanism [CEK02b]. This is not a full or particularly sophisticated encoding; however we believe it shows that the most important parts of the 2U mechanism can be encoded.

One fundamental difference between the calculus and the 2U approach is that of naming. In the standard UML-esque approach names are generally considered to be a fundamental part of the element[4] whereas in the calculus elements can be named only when they are contained within another element: the names are imposed upon them rather than being an intrinsic part. For simplicity, in our encoding we assume that elements in 2U templates are named in the same way as in our calculus; in practice this means that rather than having e.g. *class A ... end* one has *A = class ... end.*

The encoding is as follows:

[3] We recall that the $\oplus$ operator is transitive and reflexive so it makes no difference as to which order elements are selected.

[4] Although namespaces are intended to remove this difference, this assumption is still adhered to in the way class diagrams and so on are drawn.

Simple encoding of the 2U templates

$$[\![class\ n_j\ :\ C_i\ ^{i\in 1\ldots a}]\!] \triangleq [n_i\ =\ [\![C_i]\!]\ ^{i\in 1\ldots a}]$$
$$[\![Set\{A\}]\!] \triangleq SetType(A)$$
$$[\![A]\!] \triangleq Type(A)\quad \neg isSetType(A)$$
$$[\![package\ n_j\ =\ C_i\ ^{i\in 1\ldots a}]\!] \triangleq [n_i\ =\ [\![C_i]\!]\ ^{i\in 1\ldots a}]$$
$$[\![package\ (A_i\ ^{i\in 1\ldots a})\ n_j\ =\ C_j\ ^{j\in 1\ldots b}]\!] \triangleq \Lambda(A_i\ ^{i\in 1\ldots a})\ [n_j\ =\ [\![C_j]\!]\ ^{j\in 1\ldots b}]$$

The encodings for classes, packages and templates (which can only be parameterized packages in the 2U approach) are straight forward. The second and third encodings in the list are intended as simple encodings of attribute types, where we differentiate between standard and set types.

4 Future work

We feel that this paper merely scratches the surface of potential template enhancements – however we hope it gives any further enhancements a solid base to build on. Some particular features that we believe are likely to be useful range from the obvious (e.g. further control structures such as while loops) to the non-obvious (e.g. expansion within constraints); we hope this paper provides the stimulus for consideration of what features may or may not be useful. With regards to the calculus, it would be interesting to further enhance it where necessary (e.g. with integers) and also to prove some useful properties such as whether the type system is closed under reduction (and if not, to consider what changes may enable such properties to be proved).

One important feature that our calculus is unable to satisfactorily model is that of references. In object-oriented systems, there is a clear notion of a specific object A referencing another specific object B. Our calculus, effectively being a modified λ calculus, is unable to model this without involving a significant amount of extra machinery. It therefore seems natural to investigate how our approach can be sensibly integrated with **imp**ς [AC96] and **conc**ς [Han99] (which has an intuitively simple model of referencing) to achieve the 'best of both worlds.' Having references would allow us to model concepts such as inheritance and would therefore allow us to create even more powerful templates e.g. those contained within [GHJV94].

5 Conclusions

We originally motivated this paper by discussing the limitations of the current template mechanism. We have developed a typed calculus which allows one to formally express enhancements to the template mechanism and have given examples of more advanced enhancements which can be encoded in the base template calculus. Furthermore we have shown examples of usage of the enhanced template mechanism which would be impossible, or prohibitively awkward, with the standard mechanism.

The authors would like to thank Biju Appukuttan and Paul Sammut for their input.

This work was partly funded by a grant from Tata Consultancy Services.

References

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. Technical Report 7, University of Bremen, 1999.

[CEK+00] Tony Clark, Andy Evans, Stuart Kent, Steve Brodsky, and Steve Cook. A feasibility study in rearchitecting UML as a family of languages using a precise OO meta-modelling approach, 2000.

[CEK02a] Tony Clark, Andy Evans, and Stuart Kent. Engineering modelling languages: A precise meta-modelling approach. In Ralf-Detlef Kutsche and Herbert Weber, editors, *FASE 2002*, volume 2306 of *Lecture Notes in Computer Science*, 2002.

[CEK02b] Tony Clark, Andy Evans, and Stuart Kent. Revised submission to OMG RFP's ad/00-09-01 (UML 2.0 infrastructure), 2002. `http://cgi.omg.org/cgi-bin/doc?ad/02-06-14`.

[DW98] Desmond D'Souza and Alan Cameron Wills. Objects, components, and frameworks with UML : The Catalysis approach, 1998.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Orientated Software*. Addison Wesley, 1994.

[Han99] Paul Hankin. *A Study of Objects*. PhD thesis, University of Cambridge, 1999.

[OMG01] OMG. OMG unified modeling language specification 1.4, 2001. `http://www.omg.org/cgi-bin/doc?formal/01-09-67`.

[SO00] Richard Soley and OMG. Model driven architecture, 2000. `ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf`.

[TCE02] Laurence Tratt, Tony Clark, and Andy Evans. Modelling generalization and other class-to-parent relationships. Technical Report TR-02-05, Department of Computer Science, King's College London, July 2002.