$\mathcal{SDL}$—A Description Language for Building NLP Systems

Hans-Ulrich Krieger
Language Technology Lab
German Research Center for Arti£cial Intelligence (DFKI)
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
krieger@dfki.de

Abstract

We present the system description language $\mathcal{SDL}$ that offers a declarative way of specifying new complex NLP systems from already existing modules with the help of three operators: sequence, parallelism, and unrestricted iteration. Given a system description and modules that implement a minimal interface, the $\mathcal{SDL}$ compiler returns a running Java program which realizes exactly the desired behavior of the original speci£cation. The execution semantics of $\mathcal{SDL}$ is complemented by a precise formal semantics, de£ned in terms of concepts of function theory. The $\mathcal{SDL}$ compiler is part of the *SProUT* shallow language platform, a system for the development and processing of multilingual resources.

1 Introduction

In this paper, we focus on a general system description language, called $\mathcal{SDL}$, which allows the declarative speci£cation of NLP systems from a set of already existing base modules. Assuming that each initial module implements a minimal interface of methods, a new complex system is composed with the help of three operators, realizing a sequence of two modules, a (quasi-)parallel execution of several modules, and a potentially unrestricted self-application of a single module. Communication between independent modules is decoupled by a mediator which is sensitive to the operators connecting the modules and to the modules themselves. To put it in another way: new systems can be de£ned by simply putting together existing independent modules, sharing a common interface. The interface assumes functionality which modules usually already provide, such as *set input*, *clear internal state*, *start computation*, etc. It is clear that such an approach permits ¤exible experimentation with different software architectures during the set up of a new (NLP) system. The use of mediators furthermore guarantees that an independently developed module will stay independent when integrated into a new system. In the worst case, only the mediator needs to be modi£ed or upgraded, resp. In many cases, not even a modi£cation of the mediator is necessary. The execution semantics of $\mathcal{SDL}$ is complemented by an abstract semantics, de£ned in terms of concepts of function theory, such as Cartesian product, functional composition & application, Lambda abstraction, and unbounded minimization. Contrary to an interpreted approach to system speci£cation, our approach compiles a syntactically well-formed $\mathcal{SDL}$ expression into a Java program. This code might then be incorporated into a larger system or might be directly compiled by the Java compiler, resulting in an executable £le. This strategy has two advantages: £rstly, the compiled Java code is faster than an interpretation of the corresponding $\mathcal{SDL}$ expression, and secondly, the generated Java code can be modi£ed or even extended by additional software.

The structure of this paper is as follows. In the next section, we motivate the development of $\mathcal{SDL}$ and give a ¤avor of how base expressions can be compiled. We then come up with an EBNF speci£cation of the concrete syntax for $\mathcal{SDL}$ in section 3 and explain $\mathcal{SDL}$ with the help of an example. Since modules can be seen as functions in the mathematical sense, we argue in section 4 that a system speci£cation can be given a precise formal semantics. We also clarify the formal status of the mediators and show how they are incorporated in the de£nition of the abstract semantics. Section 5 then de£nes the programming interfaces and their default implementation, both for modules and for mediators. In the £nal section, we present some details of the compilation process.

2 Motivation & Idea

The shallow text processing system *SProUT* (Becker et al., 2002) developed at DFKI is a complex platform for the development and processing of multilingual resources. *SProUT* arranges processing components (e.g., tokenizer, gazetteer, named entity recognition) in a strictly sequential fashion, as is known from standard cascaded £nite-state devices (Abney, 1996).

In order to connect such (independently developed) NL components, one must look at the application programmer interface (API) of each module, hoping that there are API methods which allow, e.g., to call a module with a speci£c input, to obtain the result value, etc. In the best case, API methods from different modules can be used directly without much programming overhead. In the worst case, however, there is no API available, meaning that we have to inspect the programming code of a module and have to write additional code to realize interfaces between modules (e.g., data transformation). Even more demanding, recent hybrid NLP systems such as WHITEBOARD (Crysmann et al., 2002) implement more complex interactions and loops, instead of using a simple pipeline of modules.

We have overcome this in¤exible behavior by implementing the following idea. Since we use typed feature structures (Carpenter, 1992) in *SProUT* as the sole data interchange format between processing modules, the construction of a new system can be reduced to the interpretation of a regular expression of modules. Because the $\circ$ sign for concatenation can not be found on a keyboard, we have given the three characters $+$, $|$, and $*$ the following meaning:

- **sequence or concatenation**
  $m_1 + m_2$ expresses the fact that (1) the input to $m_1 + m_2$ is the input given to m_1, (2) the output of module m_1 serves as the input to m_2, and (3) that the £nal output of $m_1 + m_2$ is equal to the output of m_2. This is the usual ¤ow of information in a sequential cascaded shallow NL architecture.

- **concurrency or parallelism**
  $|$ denotes a quasi-parallel computation of independent modules, where the £nal output of each module serves as the input to a subsequent module (perhaps grouped in a structured object, as we do by default). This operator has far reaching potential. We envisage, e.g., the parallel computation of several morphological analyzers with different coverage or the parallel execution of a shallow topological parser and a deep HPSG parser (as in WHITEBOARD). In a programming language such as Java, the execution of modules can even be realized by independently running threads.

- **unrestricted iteration or £xpoint computation**
  m^* has the following interpretation. Module m feeds its output back into itself, until no more changes occur, thus implementing a kind of a £xpoint computation (Davey and Priestley, 1990). It is clear that such a £xpoint might not be reached in £nite time, i.e., the computation must not stop. A possible application was envisaged in (Braun, 1999), where an iterative application of a base clause module was necessary to model recursive embedding of subordinate clauses in a system for parsing German clause sentential structures. Notice that unrestricted iteration would even allow us to simulate an all-paths context-free parsing behavior, since such a feedback loop can in principle simulate an unbounded number of cascade stages in a £nite-state device (each level of a CF parse tree has been constructed by a single cascade stage).

We have de£ned a Java interface of methods which each module must ful£ll that will be incorporated in the construction of a new system. Implementing such an interface means that a module must provide an implementation for all methods speci£ed in the interface with exactly the same method name and method signature, e.g., `setInput()`, `clear()`, or `run()`. To ease this implementation, we have also implemented an abstract Java class that provides a default implementation for all these methods with the exception of `run()`, the method which starts the computation of the module and which delivers the £nal result.

The interesting point now is that a new system, declaratively speci£ed by means of the above apparatus, can be automatically compiled into a single Java class. Even the newly generated Java class implements the above interface of methods. This Java code can then be compiled by the Java compiler into a running program, realizing exactly the intended behavior of the original system speci£cation. The execution semantics for an arbitrary module `m` is de£ned to be always the execution of the `run()` method of `m`, written in Java as `m.run()`

Due to space limitations, we can only outline the basic idea and present a simpli£ed version of the compiled code for a sequence of two module instances $m_1 + m_2$, for the independent concurrent computation $m_1 \mid m_2$, and for the unbounded iteration of a single module instance m^*. Note that we use the `typewriter` font when referring to the concrete syntax or the implementation, but use *italics* to denote the abstract syntax.

$(m_1 + m_2)(\textit{input}) \equiv$

```
   m1.clear();
   m1.setInput(input);
   m1.setOutput(m1.run(m1.getInput()));
   m2.clear();
   m2.setInput(seq(m1, m2));
   m2.setOutput(m2.run(m2.getInput()));
   return m2.getOutput();
```

$(m_1 \mid m_2)(\textit{input}) \equiv$

```
   m1.clear();
   m1.setInput(input);
   m1.setOutput(m1.run(m1.getInput()));
   m2.clear();
   m2.setInput(input);
   m2.setOutput(m2.run(m2.getInput()));
   return par(m1, m2);
```

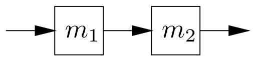
$(m^*)(\mathit{input}) \equiv$

```
   m.clear();
   m.setInput(input);
   m.setOutput(fix(m));
   return m.getOutput();
```

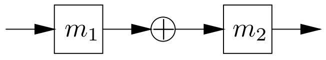
The pseudo code above contains three methods, `seq()`, `par()`, and `fix()`, methods which *mediate* between the output of one module and the input of a succeeding module. Clearly, such functionality should not be squeezed into independently developed modules, since otherwise a module m must have a notion of a £xpoint during the execution of m^* or must be sensitive to the output of every other module, e.g., during the processing of $(m_1 \mid m_2) + m$. Note that the mediators take modules as input, and so having access to their internal information via the public methods speci£ed in the module interface (the API).

The default implementation for `seq` is of course the identity function (speaking in terms of functional composition). `par` wraps the two results in a structured object (default implementation: a Java array). `fix()` implements a £xpoint computation (see section 5.3 for the Java code). These mediators can be made speci£c to special module-module combinations and are an implementation of the mediator design pattern, which loosely couples independent modules by encapsulating their interaction in a new object (Gamma et al., 1995, pp. 273). I.e., the mediators do not modify the original modules and only have read access to input and output via `getInput()` and `getOutput()`.

In the following, we present a graphical representation for displaying module combination. Given such pictures, it is easy to see where the mediators come into play. Depicting a sequence of two modules is, at £rst sight, not hard.

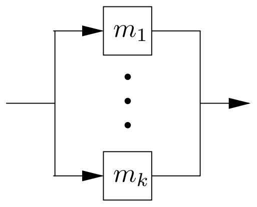


Now, if the input format of m_2 is not compatible with the output of m_1, must we change the programming code for m_2? Even more serious, if we would have another expression $m_3 + m_2$, must m_2 also be sensitive to the output format of m_3? In order to avoid these and other cases, we decouple module interaction and introduce a special mediator method for the sequence operator (`seq` in the above code), depicted by $\oplus$.

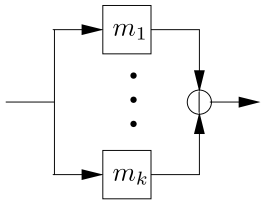


$\oplus$ connects two modules. This fact is re¤ected by making `seq` a binary method which takes `m1` and `m2` as input parameters (see example code).

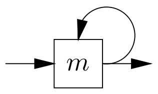
Let us now move to the parallel execution of several modules (not necessarily two, as in the above example).



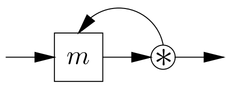
There is one problem here. What happens to the output of each module when the lines come together, meeting in the outgoing arrow? The next section has a few words on this and presents a solution. We only note here that there exists a mediator method `par`, which, by default, groups the output in a structured object. Since `par` does not know the number of modules in advance, it takes as its parameter an array of modules. Note further that the input arrows are £ne—every module gets the same data. Hence, we have the following modi£ed picture.



Now comes the $*$ operator. As we already said, the module feeds itself with its own output, until a £xpoint has been reached, i.e., until input equals output. Instead of writing



we make the mediator method for $*$ explicit, since it embodies the knowledge about £xpoints (and not the module):



3 Syntax

A new system is built from an initial set of already existing modules M with the help of the three operators $+$, $|$, and $*$. The set of all syntactically well-formed module descriptions D in $\mathcal{SDL}$ is inductively de£ned as follows:

- $m \in M \Rightarrow m \in D$
- $m_1, m_2 \in D \Rightarrow m_1 + m_2 \in D$

- $m_1, \ldots, m_k \in D \Rightarrow (|\, m_1 \ldots m_k) \in D$
- $m \in D \Rightarrow m^* \in D$

Examples in the concrete syntax are written using the typewriter font, e.g., `module`. All operators have the same priority. Succeeding modules are written from left to right, using in£x notation, e.g., `m1 + m2`.

Parallel executed modules must be put in parentheses with the `|` operator £rst, for instance `(| m1 m2)`. Note that we use the pre£x notation for the concurrency operator `|` to allow for an arbitrary number of arguments, e.g., `(| m1 m2 m3)`. This technique furthermore circumvents notorious grouping ambiguities which might lead to different results when executing the modules. Notice that since `|` must neither be commutative nor must it be associative, the result of `(| m1 m2 m3)` might be different to `(| m1 (| m2 m3))`, to `(| (| m1 m2) m3)`, or even to `(| m2 (| m1 m3))`, etc. Whether `|` is commutative or associative is determined by the implementation of concurrency mediator `par`. Let us give an example. Assume, for instance, that `m1`, `m2`, and `m3` would return typed feature structures and that `par()` would join the results by using uni£cation. In this case, `|` is clearly commutative and associative, since uni£cation is commutative and associative (and idempotent).

Finally, the unrestricted self-application of a module should be expressed in the concrete syntax by using the module name, pre£xed by the asterisk sign, and grouped using parentheses, e.g., `(* module)`. `module` here might represent a single module or a complex expression (which itself must be put in parentheses).

Making $|$ and $*$ pre£x operators (in contrast to $+$) ease the work of the syntactical analysis of an $\mathcal{SDL}$ expression. The EBNF for a complete system description *system* is given by £gure 1. A concrete running example is shown in £gure 2.

The example system from £gure 2 should be read as *de£ne a new module* `de.dfki.lt.test.System` *as* `(| rnd1 rnd2 rnd3) + inc1 + ...`, *variables* `rnd1`, `rnd2`, *and* `rnd3` *refer to instances of module* `de.dfki.lt.sdl.test.Randomize`, *module* `Randomize` *belongs to package* `de.dfki.lt.sdl.test`, *the value of* `rnd1` *should be initialized with* `("foo", "bar", "baz")`, etc. Every single line must be separated by the newline character.

The use of variables (instead of using directly module names, i.e., Java classes) has one important advantage: variables can be reused (viz., `rnd2` and `rnd3` in the example), meaning that the same instances are used at several places throughout the system description, instead of using several instances of the same module (which, of course, can also be achieved; cf. `rnd1`, `rnd2`, and `rnd3` which are instances of module `Randomize`). Notice that the value of a variable can not be rede£ned during the course of a system description.

4 Modules as Functions

Before we start the description of the implementation in the next section, we will argue that a system description can be given a precise formal semantics, assuming that the initial modules, which we call *base modules* are well de£ned. First of all, we only need some basic mathematical knowledge from secondary school, viz., the concept of a function.

A function f (sometimes called a mapping) from S to T, written as $f : S \longrightarrow T$, can be seen as a special kind of relation, where the domain of f is S (written as $\textsc{dom}(f) = S$), and for each element in the domain of f, there is at most one element in the range (or codomain) $\textsc{rng}(f)$. If there always exists an element in the range, we say that f is a total function (or well de£ned) and write $f\downarrow$. Otherwise, f is said to be a partial function, and for an $s \in S$ for which f is not de£ned, we then write $f(s)\uparrow$.

Since S itself might consist of ordered n-tuples and thus is the Cartesian product of $S_1, \ldots, S_n$, depicted as $\times_{i=1}^{n} S_i$, we use the vector notation and write $f(\vec{s})$ instead of $f(s)$. The n-fold functional composition of $f : S \longrightarrow T$ ($n \geq 0$) is written as f^n and has the following inductive de£nition: $f^0(\vec{s}) := \vec{s}$ and $f^{i+1}(\vec{s}) := f(f^i(\vec{s}))$.

$s \in S$ is said to be a *£xpoint* of $f : S \longrightarrow S$ iff $f(f(s)) =_S f(s)$ (we use $=_S$ to denote the equality relation in S).

Assuming that `m` is a module for which a proper `run()` method has been de£ned, we will, from now on, refer to the function m as abbreviating `m.run()`, the execution of method `run()` from module `m`. Hence, we de£ne the execution semantics of m to be equivalent to `m.run()`.

4.1 Sequence

Let us start with the sequence $m_1 + m_2$ of two modules, regarded as two function $m_1 : S_1 \longrightarrow T_1$ and $m_2 : S_2 \longrightarrow T_2$. $+$ here is the analogue to functional composition $\circ$, and so we de£ne the meaning (or abstract semantics) $[\![\cdot]\!]$ of $m_1 + m_2$ as

$$[\![m_1 + m_2]\!](\vec{s}) := (m_2 \circ m_1)(\vec{s}) = m_2(m_1(\vec{s}))$$

$m_1 + m_2$ then is well-de£ned if $m_1\downarrow$, $m_2\downarrow$, and $T_1 \subseteq S_2$ is the case, due to the following biconditional:

$$m_1\downarrow, m_2\downarrow,\ T_1 \subseteq S_2 \Longleftrightarrow (m_1 \circ m_2 \colon S_1 \longrightarrow T_2)\downarrow$$

4.2 Parallelism

We now come to the parallel execution of k modules $m_i : S_i \longrightarrow T_i$ ($1 \leq i \leq k$), operating on the same input. As already said, the default mediator for $|$ returns an ordered

system	$\longrightarrow$ *de£nition* {*command*}* *variables*
de£nition	$\longrightarrow$ *module* `"="` *regexpr newline*
module	$\longrightarrow$ a fully quali£ed Java class name
regexpr	$\longrightarrow$ *var* | `"("` *regexpr* `")"` | *regexpr* `"+"` *regexpr* | `"("` `"|"` {*regexpr*}$^+$ `")"` | `"("` `"*"` *regexpr* `")"`
newline	$\longrightarrow$ the newline character
command	$\longrightarrow$ *mediator* | *threaded*
mediator	$\longrightarrow$ `"Mediator ="` *med newline*
med	$\longrightarrow$ a fully quali£ed Java class name
threaded	$\longrightarrow$ `"Threaded ="` {`"yes"` | `"no"`} *newline*
variables	$\longrightarrow$ {*vareq newline*}$^+$
vareq	$\longrightarrow$ *var* `"="` *module* [*initexpr*]
var	$\longrightarrow$ a lowercase symbol
initexpr	$\longrightarrow$ `"("` *string* {`","` *string*}* `")"`
string	$\longrightarrow$ a Java string

Figure 1: The EBNF for the syntax of $\mathcal{SDL}$.

```
de.dfki.lt.test.System = (| rnd1 rnd2 rnd3) + inc1 + inc2 + (* i5ut42) + (* (rnd3 + rnd2))
Mediator = de.dfki.lt.sdl.test.MaxMediator
Threaded = Yes
rnd1 = de.dfki.lt.sdl.test.Randomize("foo", "bar", "baz")
rnd2 = Randomize("bar", "baz")
rnd3 = de.dfki.lt.sdl.test.Randomize("baz")
inc1 = de.dfki.lt.sdl.test.Increment
inc2 = de.dfki.lt.sdl.test.Increment
i5ut42 = de.dfki.lt.sdl.test.Incr5UpTo42
```

Figure 2: An example in the concrete syntax of $\mathcal{SDL}$.

sequence of the results of $m_1, \ldots, m_k$, hence is similar to the Cartesian product $\times$:

$$[\![(|\ m_1\ \ldots\ m_k)]\!](\vec{s}) := \langle\, m_1(\vec{s}), \ldots, m_k(\vec{s})\,\rangle$$

$(|\ m_1\ \ldots\ m_k)$ is well-de£ned if each module is well de£ned and the domain of each module is a superset of the domain of the new composite module:

$$m_1\!\downarrow, \ldots, m_k\!\downarrow \Longrightarrow$$

$$(m_1 \times \ldots \times m_k : (S_1 \cap \ldots \cap S_k)^k \longrightarrow T_1 \times \ldots \times T_k)\!\downarrow$$

4.3 Iteration

A proper de£nition of unrestricted iteration, however, deserves more attention and a bit more work. Since a module m feeds its output back into itself, it is clear that the iteration $(m^*)(\vec{s})$ must not terminate. I.e., the question whether $m^*\!\downarrow$ holds, is undecidable in general. Obviously, a necessary condition for $m^*\!\downarrow$ is that $S \supseteq T$, and so if $m : S \longrightarrow T$ and $m\!\downarrow$ holds, we have $m^* : S \longrightarrow S$. Since m is usually not a monotonic function, it must not be the case that m has a least and a greatest £xpoint. Of course, m might not possess any £xpoint at all.

Within our very practical context, we are interested in £nitely-reachable £xpoints. From the above remarks, it is clear that given $\vec{s} \in S$, $(m^*)(\vec{s})$ terminates in £nite time iff no more changes occur during the iteration process, i.e.,

$$\exists n \in \mathbf{N} \,.\, m^n(\vec{s}) =_S m^{n-1}(\vec{s})$$

We can formalize the meaning of $*$ with the help of Kleene's μ operator, known from recursive function theory (Hermes, 1978). μ is a functional and so, given a function f as its input, returns a new function $\mu(f)$, the unbounded minimization of f. Originally employed to precisely de£ne (partial) recursive functions of natural numbers, we need a slight generalization, so that we can apply μ to functions, not necessarily operating on natural numbers.

Let $f : \mathbf{N}^{k+1} \longrightarrow \mathbf{N}$ ($k \in \mathbf{N}$). $\mu(f) : \mathbf{N}^k \longrightarrow \mathbf{N}$ is given by

$$\mu(f)(\vec{x}) := \begin{cases} n & \text{if } f(\vec{x}, n) = 0 \text{ and } f(\vec{x}, i) > 0, \\ & \text{for all } 0 \leq i \leq n-1 \\ \uparrow & \text{otherwise} \end{cases}$$

I.e., $\mu(f)(\vec{x})$ returns the least n for which $f(\vec{x}, n) = 0$. Such an n, of course, must not exist.

We now move from the natural numbers $\mathbf{N}$ to an arbitrary (structured) set S with equality relation $=_S$. The task of μ here is to return the number of iteration steps

n for which a self-application of module m no longer changes the output, when applied to the original input $\vec{s} \in S$. And so, we have the following de£nitional equation for the meaning of m^*:

$$[\![m^*]\!](\vec{s}) := m^{\mu(m)(\vec{s})}(\vec{s})$$

Obviously, the number of iteration steps needed to obtain a £xpoint is given by $\mu(m)(\vec{s})$, where $\mu : (S \longrightarrow S) \longrightarrow \mathbf{N}$. Given m, we de£ne $\mu(m)$ as

$$\mu(m)(\vec{s}) := \begin{cases} n & \text{if } m^n(\vec{s}) =_S m^{n-1}(\vec{s}) \text{ and} \\ & m^i(\vec{s}) \neq_S m^{i-1}(\vec{s}), \\ & \text{for all } 0 \leq i \leq n-1 \\ \uparrow & \text{otherwise} \end{cases}$$

Compare this de£nition with the original $\mu(f)(\vec{x})$ on natural numbers above. Testing for zero is replaced here by testing for equality in S. This last de£nition completes the semantics for m^*.

4.4 Incorporating Mediators

The above formalization does not include the use of mediators. The effects the mediators have on the input/output of modules are an integral part of the de£nition for the meaning of $m_1 + m_2$, $(|\ m_1 \ \ldots \ m_k)$, and m^*. In case we explicitly want to represent (the default implementation of) the mediators in the above de£nitions, we must, £rst of all, clarify their status.

Let us focus, for instance, on the mediator for the sequence operator $+$. We already said that the mediator for $+$ uses the output of m_1 to feed m_2, thus can be seen as the identity function *id*, speaking in terms of functional composition. Hence, we might rede£ne $[\![(m_1 + m_2)]\!](\vec{s})$ as

$$[\![(m_1 + m_2)]\!](\vec{s}) :=$$

$$(m_2 \circ \mathit{id} \circ m_1)(\vec{s}) = m_2(\mathit{id}(m_1(\vec{s}))) = m_2(m_1(\vec{s}))$$

If so, mediators were functions and would have the same status as modules. Clearly, they pragmatically differ from modules in that they coordinate the interaction between independent modules (remember the mediator metaphor). However, we have also said that the mediator methods take modules as input. When adopting this view, a mediator is different from a module: it is a functional (as is μ), taking functions as arguments (the modules) and returning a function. Now, let $\mathcal{S}$ be the mediator for the $+$ operator. We then obtain a different semantics for $m_1 + m_2$.

$$[\![(m_1 + m_2)]\!](\vec{s}) := (m_2 \circ \mathcal{S}(m_1, m_2) \circ m_1)(\vec{s})$$

and

$$\mathcal{S}(m_1, m_2) := \mathit{id}$$

is the case in the default implementation for $+$. This view, in fact, precisely corresponds to the implementation.

Let us quickly make the two other de£nitions re¤ect this new view and let $\mathcal{P}$ and $\mathcal{F}$ be the functionals for $|$ and $*$, resp. For $|$, we now have

$$[\![(|\ m_1 \ \ldots \ m_k)]\!](\vec{s}) := (\mathcal{P}(m_1, \ldots, m_k) \circ (\times_{i=1}^k m_i))(\vec{s}^{\,k})$$

$(\times_{i=1}^k m_i)(\vec{s}^{\,k})$ denotes the ordered sequence $\langle\, m_1(\vec{s}), \ldots, m_k(\vec{s})\,\rangle$ to which function $\mathcal{P}(m_1, \ldots, m_k)$ is applied. At the moment,

$$\mathcal{P}(m_1, \ldots, m_k) := \times_{i=1}^k \mathit{id}$$

i.e., the identity function is applied to the result of each $m_i(\vec{s})$, and so in the end, we still obtain $\langle\, m_1(\vec{s}), \ldots, m_k(\vec{s})\,\rangle$.

The adaption of m^* is also not hard: $\mathcal{F}$ is exactly the $\mu(m)(\vec{x})$-fold composition of m, given value $\vec{x}$. Since $\vec{x}$ are free variables, we use Church's Lambda abstraction (Barendregt, 1984), make them bound, and write

$$\mathcal{F}(m) := \lambda\vec{x} \,.\, m^{\mu(m)(\vec{x})}(\vec{x})$$

Thus

$$[\![m^*]\!](\vec{s}) := (\mathcal{F}(m))(\vec{s})$$

It is clear that the above set of de£nitions is still not complete, since it does not cover the cases where a module m consists of several submodules, as does the syntax of $\mathcal{SDL}$ clearly admit. This leads us to the £nal four inductive de£nitions which conclude this section:

- $[\![m]\!](\vec{s}) := m(\vec{s})$ iff m is a base module
- $[\![(m_1 + m_2)]\!](\vec{s}) :=$
  $([\![m_2]\!] \circ \mathcal{S}([\![m_1]\!], [\![m_2]\!]) \circ [\![m_1]\!])(\vec{s})$
- $[\![(|\ m_1 \ \ldots \ m_k)]\!](\vec{s}) :=$
  $(\mathcal{P}([\![m_1]\!], \ldots, [\![m_k]\!]) \circ (\times_{i=1}^k [\![m_i]\!]))(\vec{s}^{\,k})$
- $[\![m^*]\!](\vec{s}) := (\mathcal{F}([\![m]\!]))(\vec{s})$,
  whereas $\mathcal{F}([\![m]\!]) := \lambda\vec{x} \,.\, [\![m]\!]^{\mu([\![m]\!])(\vec{x})}(\vec{x})$

Recall that the execution semantics of $m(\vec{s})$ has not changed after all and is still `m.run(s)`, whereas `s` abbreviates the Java notation for the k-tuple $\vec{s}$.

5 Interfaces

This section gives a short scetch of the API methods which every module must implement and presents the default implementation of the mediator methods.

5.1 Module Interface `IModule`

The following seven methods must be implemented by a module which should contribute to a new system. The next subsection provides a default implementation for six of them. The exception is the one-argument method `run()` which is assumed to execute a module.

- `clear()` clears the internal state of the module it is applied to. `clear()` is useful when a module instance is reused during the execution of a system. `clear()` might throw a `ModuleClearError` in case something goes wrong during the clearing phase.
- `init()` initializes a given module by providing an array of init strings. `init()` might throw a `ModuleInitError`.
- `run()` starts the execution of the module to which it belongs and returns the result of this computation. An implementation of `run()` might throw a `ModuleRunError`. Note that `run()` should not store the input nor the output of the computation. This is supposed to be done independently by using `setInput()` and `setOutput()` (see below).
- `setInput()` stores the value of parameter `input` and returns this value.
- `getInput()` returns the input originally given to `setInput()`.
- `setOutput()` stores the value of parameter `output` and returns this value.
- `getOutput()` returns the output originally given to `setOutput()`.

5.2 Module Methods

Six of the seven module methods are provided by a default implementation in class `Modules` which implements interface `IModule` (see above). New modules are advised to inherit from `Modules`, so that only `run()` must actually be speci£ed. Input and output of a module is memorized by introducing the two additional private instance £elds `input` and `output`.

```
public abstract class Modules implements IModule {

  private Object input, output;

  protected Modules() {
    this.input = null;
    this.output = null; }

  public Object run(Object input) throws
    UnsupportedOperationException {
    throw new UnsupportedOperationException("..."); }

  public void clear() {
    this.input = null;
    this.output = null; }

  public void init(String[] initArgs) {
  }

  public Object setInput(Object input) {
    return (this.input = input); }

  public Object getInput() {
    return this.input; }

  public Object setOutput(Object output) {
    return (this.output = output); }

  public Object getOutput() {
    return this.output; }

}
```

5.3 Mediator Methods

The public class `Mediators` provides a default implementation for the three mediator methods, speci£ed in interface `IMediator`. It is worth noting that although `fix()` returns the £xpoint, it relocates its computation into an auxiliary method `fixpoint()` (see below), due to the fact that mediators are not allowed to change the internal state of a module. And thus, the input £eld still contains the original input, whereas the output £eld refers to the £xpoint, at last.

```
public class Mediators implements IMediator {

  public Mediators() {
  }

  public Object seq(IModule module1, IModule module2) {
    return module1.getOutput(); }

  public Object par(IModule[] modules) {
    Object[] result = new Object[modules.length];
    for (int i = 0; i < modules.length; i++)
      result[i] = modules[i].getOutput();
    return result; }

  public Object fix(IModule module) {
    return fixpoint(module, module.getInput()); }

  private Object fixpoint(IModule module, Object input) {
    Object output = module.run(input);
    if (output.equals(input))
      return output;
    else
      return fixpoint(module, output); }

}
```

6 Compiler

In section 2, we have already seen how basic expressions are compiled into a sequence of instructions, consisting of API methods from the module and mediator interface. Here, we like to glance at the compilation of more complex $\mathcal{SDL}$ expressions.

First of all, we note that complex expressions are decomposed into ¤at basic expressions which are not further structured. Each subexpression is associated with a new module variable and these variables are inserted into the original system description which will also then become ¤at. In case of the example from £gure 2, we have the following subexpressions together with their variables (we pre£x every variable by the dollar sign): `$1 = (| $rnd1 $rnd2 $rnd3)`, `$2 = (* $i5ut42)`, `$3 = ($rnd3 + $rnd2)`, and `$4 = (* $3)`. As a result, the original system description reduces to `$1 + $inc1 + $inc2 + $2 + $4` and thus is normalized as `$1`, . . ., `$4` are. The 𝒮𝒟ℒ compiler then introduces so-called *local* or *inner Java classes* for such subexpressions and locates them in the same package to which the newly de£ned system belongs. Clearly, each new inner class must also ful£ll the module interface `IModule` (see section 5) and the 𝒮𝒟ℒ compiler produces the corresponding Java code, similar to the default implementation in class `Modules` (section 5), together with the right constructors for the inner classes.

For each base module and each newly introduced inner class, the compiler generates a private instance £eld (e.g., `private Randomize $rnd1`) and a new instance (e.g., `this.$rnd1 = new Randomize()`) to which the API methods can be applied. Each occurence of the operators `+`, `|`, and `*` corresponds to the execution of the mediator methods `seq`, `par`, and `fix` (see below).

Local variables (pre£xed by the low line character) are also introduced for the individual `run()` methods (`_15`, . . ., `_23` below). These variables are introduced by the 𝒮𝒟ℒ compiler to serve as handles (or anchors) to already evaluated subexpression, helping to establish a proper ¤ow of control during the recursive compilation process.

We £nish this paper by presenting the generated code for the `run()` method for system `System` from £gure 2.

```
public Object run(Object input)
  throws ModuleClearError, ModuleRunError {
  this.clear();
  this.setInput(input);
  IMediator _med = new MaxMediator();
  this.$1.clear();
  this.$1.setInput(input);
  Object _15 = this.$1.run(input);
  this.$1.setOutput(_15);
  Object _16 = _med.seq(this.$1, this.$inc1);
  this.$inc1.clear();
  this.$inc1.setInput(_16);
  Object _17 = this.$inc1.run(_16);
  this.$inc1.setOutput(_17);
  Object _18 = _med.seq(this.$inc1, this.$inc2);
  this.$inc2.clear();
  this.$inc2.setInput(_18);
  Object _19 = this.$inc2.run(_18);
  this.$inc2.setOutput(_19);
  Object _20 = _med.seq(this.$inc2, this.$2);
  this.$2.clear();
  this.$2.setInput(_20);
  Object _21 = this.$2.run(_20);
  this.$2.setOutput(_21);
  Object _22 = _med.seq(this.$2, this.$4);
  this.$4.clear();
  this.$4.setInput(_22);
  Object _23 = this.$4.run(_22);
  this.$4.setOutput(_23);
  return this.setOutput(_23);
  }
```

We always generate a new mediator object (`_med`) for each local class in order to make the parallel execution of modules thread-safe. Note that in the above code, the mediator method `seq()` is applied four times due to the fact that $+$ occurs four times in the original speci£cation.

The full code generated by the 𝒮𝒟ℒ compiler for the example from £gure 2 can be found under `http://www.dfki.de/∼krieger/public/`. The directory also contains the Java code of the involved modules, plus the default implementation of the mediator and module methods. In the workshop, we hope to further report on the combination of WHAT (Schäfer, 2003), an XSLT-based annotation transformer, with 𝒮𝒟ℒ.

Acknowledgement

I am grateful to my colleagues Bernd Kiefer, Markus Pilzecker, and Ulrich Schäfer, helping me to make things clear. Thanks to the anonymous reviewers who have identi£ed weak points. This work was supported by the German Federal Ministry for Education, Science, Research, and Technology under grant no. 01 IW C02 (QUETAL) and by an EU grant under no. IST 12179 (Airforce).

References

S. Abney. 1996. Partial parsing via £nite-state cascades. *Natural Language Engineering*, 2(4):337–344.

H. Barendregt. 1984. *The Lambda Calculus, its Syntax and Semantics*. North-Holland.

M. Becker, W. Dro££dzyński, H.-U. Krieger, J. Piskorski, U. Schäfer, and F. Xu. 2002. SProUT—shallow processing with uni£cation and typed feature structures. In *Proceedings of ICON*.

C. Braun. 1999. Flaches und robustes Parsen Deutscher Satzgefüge. Master's thesis, Universität des Saarlandes. In German.

B. Carpenter. 1992. *The Logic of Typed Feature Structures*. Cambridge University Press.

B. Crysmann, A. Frank, B. Kiefer, S. Müller, G. Neumann, J. Piskorski, U. Schäfer, M. Siegel, H. Uszkoreit, F. Xu, M. Becker, and H.-U. Krieger. 2002. An integrated architecture for shallow and deep processing. In *Proceedings of ACL*, pages 441–448.

B.A. Davey and H.A. Priestley. 1990. *Introduction to Lattices and Order*. Cambridge University Press.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley.

H. Hermes. 1978. *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit: Einführung in die Theorie der rekursiven Funktionen*. Springer, 3rd ed. In German. Also as *Enumerability, Decidability, Computability: An Introduction to the Theory of Recursive Functions*.

U. Schäfer. 2003. WHAT: an XSLT-based infrastructure for the integration of natural language processing components. In *Proceedings of SEALTS*.