**Deutsches Forschungszentrum für Künstliche Intelligenz GmbH**

**Research Report**
RR-03-01

# SDL—A System Description Language

**Hans-Ulrich Krieger**

**March 2003**

**Deutsches Forschungszentrum für Künstliche Intelligenz GmbH**

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210
E-Mail: info@dfki.uni-kl.de

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341
E-Mail: info@dfki.de

WWW: http://www.dfki.de

## Deutsches Forschungszentrum für Künstliche Intelligenz

# DFKI GmbH

## German Research Center for Artificial Intelligence

Founded in 1988, DFKI today is one of the largest nonprofit contract research institutes in the field of innovative software technology based on Artificial Intelligence (AI) methods. DFKI is focusing on the complete cycle of innovation — from world-class basic research and technology development through leading-edge demonstrators and prototypes to product functions and commercialization.

Based in Kaiserslautern and Saarbrücken, the German Research Center for Artificial Intelligence ranks among the important “Centers of Excellence” worldwide.

An important element of DFKI’s mission is to move innovations as quickly as possible from the lab into the marketplace. Only by maintaining research projects at the forefront of science can DFKI have the strength to meet its technology transfer goals.

DFKI has about 165 full-time employees, including 141 research scientists with advanced degrees. There are also around 95 part-time research assistants.

Revenues for DFKI were about 30 million DM in 2000, half from government contract work and half from commercial clients. The annual increase in contracts from commercial clients was greater than 20% during the last three years.

At DFKI, all work is organized in the form of clearly focused research or development projects with planned deliverables, various milestones, and a duration from several months up to three years.

DFKI benefits from interaction with the faculty of the Universities of Saarbrücken and Kaiserslautern and in turn provides opportunities for research and Ph.D. thesis supervision to students from these universities, which have an outstanding reputation in Computer Science.

The key directors of DFKI are Prof. Wolfgang Wahlster (CEO) and Dr. Walter Olthoff (CFO).

DFKI’s five research departments are directed by internationally recognized research scientists:

- Knowledge Management (Director: Prof. A. Dengel)
- Intelligent Visualization and Simulation Systems (Director: Prof. H. Hagen)
- Deduction and Multiagent Systems (Director: Prof. J. Siekmann)
- Language Technology (Director: Prof. H. Uszkoreit)
- Intelligent User Interfaces (Director: Prof. W. Wahlster)

In this series, DFKI publishes research reports, technical memos, documents (eg. workshop proceedings), and final project reports. The aim is to make new results, ideas, and software available as quickly as possible.

Prof. Wolfgang Wahlster
Director

# *SDL*—A System Description Language

**Hans-Ulrich Krieger**

DFKI-RR-03-01

I am grateful to my colleagues from the *SProUT* group, viz., Markus Becker, Witold Drożdżyński, Jakub Piskorski, Ulrich Schäfer, and Feiyu Xu who provide an excellent environment for testing *SDL*. The discussions with Bernd Kiefer, Markus Pilzecker, and Jörg Steffen have helped me to make things clear—thank you guys! The feedback from Hans Uszkoreit and Anette Frank made me realize that there is a need for such a specification language. The following people have read earlier versions of this report and I greatly appreciate their comments: Mark-Jan Nederhof, Hans Uszkoreit, and Feiyu Xu. Parts of this work were supported by the German Federal Ministry for Education, Science, Research, and Technology under grant no. 01 IN A01 (Collate) and by an EU grant under no. IST 12179 (Airforce).



ISSN 0946-008X

# $\mathcal{SDL}$—A System Description Language

Hans-Ulrich Krieger
German Research Center for Artificial 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 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 specification. The execution semantics of $\mathcal{SDL}$ is complemented by a precise formal semantics, defined 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. We believe that the application of $\mathcal{SDL}$ is not only limited to the construction of pure NLP systems, but can also be employed in the definition of general software systems.

# 1 Introduction

In this report, we motivate the development of a general system description language, called $\mathcal{SDL}$, which allows the declarative specification of software systems from a set of already existing modules. Assuming that each initial module implements a minimal interface of methods, a new complex system is composed with the help of three operators that realize 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.
This means that new complex systems can be defined 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 allows to flexibly experiment 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 modified or upgraded, resp. In many cases, not even a modification of the mediator is necessary.
Contrary to an interpreted approach to system specification, our approach compiles a syntactically well-formed $\mathcal{SDL}$ expression into a Java program (aka a Java class). This code might then be incorporated into a larger system or might be directly compiled by the Java compiler, resulting in an executable file. This strategy has two advantages: firstly, the compiled Java code is faster than an interpretation of the corresponding $\mathcal{SDL}$ expression, and secondly, the generated Java code can be modified or even extended by additional software.
The structure of this report is as follows. In the next section, we motivate the development of $\mathcal{SDL}$ and give a flavor of how the implementation is realized. We then come up with an EBNF specification of the concrete syntax for $\mathcal{SDL}$ and explain $\mathcal{SDL}$ with the help of an example. Since modules can be seen as functions in the mathematical sense, we argue that a system specification can be given a formal semantics. The following section specifies the interface every single module must fulfill. The section also specifies the mediator interface. After that, we inspect the default implementation for the module methods, look at the mediator code, and finally describe how the $\mathcal{SDL}$ compiler transforms arbitrary $\mathcal{SDL}$ expressions into a set of Java classes. We close this report by putting our approach into a larger context and by indicating our next steps. The final appendix presents a running example of a system of simple modules which embodies a complex flow of control.

# 2 Motivation

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 finite-state devices (e.g., Abney 1996 or Hobbs et al. 1997).
In order to connect such 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 specific 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 (e.g., Crysmann et al. 2002) implement more complex interactions and loops instead of using a simple pipeline of modules.
We have overcome this inflexible 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 final output of $m_1 + m_2$ is equal to the output of $m_2$. This is the usual flow of information in a sequential cascaded shallow architecture.

- **concurrency or parallelism**
  $|$ denotes a quasi-parallel computation of independent modules, where the final output of each module serves as the input to a subsequent module (perhaps grouped in a structured object; we do this!). This operator has a far reaching potential. We envisage, e.g., the parallel computation of several morphological analyzers with different coverage. In a programming language such as Java (Arnold et al. 2000), the execution of modules can even be realized by independently running threads (see next section on how to specify this).

- **unrestricted iteration or fixpoint 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 fixpoint computation (Davey and Priestley 1990). It is clear that such a fixpoint might not be reached in finite 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. Unrestricted iteration would even allow us to simulate an all-paths context-free parsing behavior, since such a feedback loop can in principle realize an unbounded number of cascade stages in a finite-state device.

We have defined a Java interface of methods which each module must fulfill that will be incorporated in the construction of a new system (see section 5). Implementing such an interface means that a module must provide an implementation for all methods specified 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 final result.
The interesting point now is that a new system, declaratively specified by means of the above apparatus, can be automatically compiled into a single Java program, i.e., a Java class (see section 6). This Java code can then be compiled by the Java compiler into a running program (a class file for Java's VM) which realizes exactly the intended behavior of the original system specification. Obtaining a module instance from a module name (i.e., a Java class) is achieved by importing the corresponding Java class in the compiled code and by applying `new` to the class name. Contrary to the compilation method described here, an interpreted approach to system description must refer to a new module at run time through Java's reflection API by using `forName()` and `newInstance()` which allows for dynamic loading of classes into the Java virtual machine (Arnold et al. 2000).
At this point, we can only outline the basic idea and present a simplified 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 $m^*$. The notation `m.method()` is the Java notation for executing the method `method()` from class (or as we say here: module) `m`.

$(m_1 + m_2)(input) \equiv$

```
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();
```

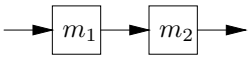
$(m_1 \mid m_2)(input) \equiv$

```
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);
```

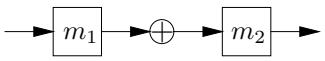
$(m^*)(input) \equiv$

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

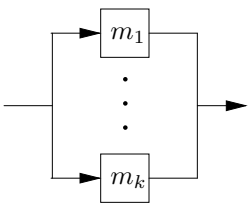
The pseudo code above contains three methods, `seq()`, `par()`, and `fix()`, methods which mediate between the output of one module and the input of a succeeding module. Clearly, such functionality should not be squeezed into independently developed modules, since otherwise a module $m$ must have a notion of a fixpoint 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 specified in the module interface.
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 fixpoint computation. These mediators can be made specific to special module-module combinations and are an implementation of the mediator pattern, which loosely couples independent modules by encapsulating their interaction in a new object; see Gamma et al. 1995, pp. 273. I.e., these mediators do not modify the original modules and only have read access to input and output via `getInput()` and `getOutput()`.
The need for the mediators originally arose when we tried to display module combinations as pictures. Furthermore, from the standpoint of symmetry, we thought that for each operator a corresponding mediator method could be useful, and so must exist. Let us give a few examples to see how this basic idea progressed.
Depicting a sequence of two modules is, at first sight, not hard.



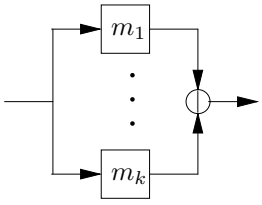

Now, if the input format of $m_2$ is not compatible with the output of $m_1$, must we change the programming code for $m_2$? Even more serious, if we would have another expression $m_3 + m_2$, must $m_2$ also be sensitive to the output format of $m_3$? In order to avoid these and other cases, we decouple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



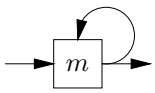

$\oplus$ connects two modules. This fact is reflected by making `seq` a binary method which takes `m1` and `m2` as input parameters (see example code).
Let us now move to the parallel execution of several modules (not necessarily two, as in the above example).



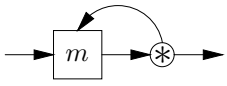
There is one problem here. What happens to the output of each module when the lines come together, meeting in the outgoing arrow? The next section has a few words on this and presents a solution. We only note here that there exists a mediator method `par`, which, by default, groups the output in a structured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




Now comes the $*$ operator. As we already said, the module feeds itself with its own output, until a fixpoint has been reached. Instead of writing




we make the mediator method for $*$ explicit, since it embodies the knowledge about fixpoints (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 defined as follows:

- $m \in M \Rightarrow m \in D$
- $m_1, m_2 \in D \Rightarrow m_1 + m_2 \in D$
- $m_1, \ldots, m_k \in D \Rightarrow (\mid m_1 \ldots m_k) \in D$
- $m \in D \Rightarrow (* \; m) \in D$

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 infix notation, e.g., `m1 + m2`.
Parallel executed modules must be put in parentheses with the `|` operator first, for instance `(| m1 m2)`. Note that we use the prefix 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` (see last section for the motivation). 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 unification. In this case, `|` is clearly commutative and associative, since unification is commutative and associative (and idempotent).
Finally, the unrestricted self-application of a module should be expressed by using the module name, prefixed 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 $*$ prefix 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 figure 1.

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

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

A concrete, although unrealistic example is shown in figure 2.
We have tried to define a syntax which employs only a small number of reserved words, viz., `=`, `(`, `)`, `+`, `|`, and `*`. The example system from figure 2 should be read as *define a new module* `de.dfki.lt.system.S` *as* `a + b + ...`, *variables* `a`, `b`, *and* `d` *refer to instances of module* `de.dfki.lt.system.A`, *module* `de.dfki.lt.system.A` *belongs to package* `de.dfki.lt.system`, *the value of* `c` *should be initialized with* `("foo", "bar", "baz")`, etc. Every single line must be separated by the newline character.

```
de.dfki.lt.system.S = a + b + (| (c + a) d) + d + (* (a + b)) + e
a = de.dfki.lt.system.A
b = de.dfki.lt.system.A
c = C ("foo", "bar", "baz")
d = de.dfki.lt.system.A
e = de.dfki.lt.system.test.E ("../test/corpus.txt")
```

Figure 2: An example in the concrete syntax of *SDL*.

Note that the package specification and the initialization arguments are optional (see figure 1). In our case, packages are specified for module `A` and `E`. Module `C` thus belongs to what Java calls the *unnamed* (or sometimes the default) package (Arnold et al. 2000, p. 32). Furthermore, only `c` and `e` are initialized, using method `init()` from the module interface (see next section).
The use of variables (instead of using directly module names = Java classes) has one important advantage: variables can be reused (e.g., `a` in the example), meaning that the same instance is 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. `a`, `b`, and `d` which are instances of module `de.dfki.lt.system.A`). Notice that the value of a variable can not be redefined during the course of a system description.
The above example as well as the EBNF of *SDL* make use of the Java package concept. We assume a basic familiarity with this fundamental notion; see, e.g., Arnold et al. 2000 or Flanagan 2002.

# 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 defined. 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 $\text{DOM}(f) = S$), and for each element in the domain of $f$, there is at most one element in the range (or codomain) $\text{RNG}(f)$.
If there always exists an element in the range, we say that $f$ is a total function (or well defined) and write $f\downarrow$. Otherwise, $f$ is said to be a partial function, and for an $s \in S$ for which $f$ is not defined, 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 definition: $f^0(\vec{s}) := \vec{s}$ and $f^{i+1}(\vec{s}) := f(f^i(\vec{s}))$.
$s \in S$ is said to be a *fixpoint* 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 defined, we will, from now on, refer to the function $m$ as abbreviating `m.run()`, the execution of method `run()` from module `m`. Hence, we define 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 define 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-defined if $m_1\downarrow$, $m_2\downarrow$, and $T_1 \subseteq S_2$ is the case, due to the following biconditional:

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

## 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 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-defined if each module is well defined 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 definition 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 fixpoint. Of course, $m$ might not possess any fixpoint at all.
Within our very practical context, we are interested in finitely-reachable fixpoints. From the above remarks, it is clear that given $\vec{s} \in S$, $(*\ m)(\vec{s})$ terminates in finite 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 the $\mu$ operator, known from recursive function theory and introduced by Stephen C. Kleene in 1936 (Hermes 1978). $\mu$ 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 define (partial) recursive functions of natural numbers, we need a slight generalization, so that we can apply $\mu$ to functions, not necessarily operating on natural numbers. We start with the original formulation and then adapt this framework to our needs.
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 $\mu$ 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 definitional 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 fixpoint is given by $\mu(m)(\vec{s})$, where $\mu : (S \longrightarrow S) \longrightarrow \mathbf{N}$. Given $m$, we define $\mu(m)$ 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 definition 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 definition 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 definition 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 definitions, we must, first 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 redefine $[\![(m_1 + m_2)]\!](\vec{s})$ as

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

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, e.g., $\mu$), 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 $m_1 + m_2$.

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

where

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

is the case in the default implementation for +. This view, in fact, precisely corresponds to the implementation.
Let us quickly make the two other definitions reflect this new view and let $\mathcal{P}$ and $\mathcal{F}$ be the functionals for | and $*$, resp. For |, we now have

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

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

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

i.e., the identity function is applied to the result of each $m_i(s_i)$, 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 definitions 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 (see example in the appendix). This leads us to the final four inductive definitions 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 deals with interfaces, the Java `IModule` interface which each module must fulfill, and the Java `IMediator` interface that enforces methods for each system-building operator. In addition, we repeat the standard Java interface `Runnable` due to the fact that we allow concurrently executed modules to run as true Java threads. Note that we assume a basic familiarity with the syntax of Java and its exception/error concept.

## 5.1 Module Interface `IModule`

The following seven methods must be implemented by a module which should contribute to a new system. Section 6 provides a default implementation for six of them. The exception is the one-argument method `run()` which is assumed to execute a module.

- `public void clear() throws ModuleClearError;`
  `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 (see, e.g., variable `a` in figure 2). In the simplified compiled code earlier, `clear()` is always executed before the input to a module is stored. `clear()` might throw a `ModuleClearError` in case something goes wrong during the clearing phase.

- `public void init(String[] initArgs) throws ModuleInitError;`
  `init()` initializes a given module by providing an array of init strings. The instances of module `de.dfki.lt.system.test.E` and `C` in figure 2 are examples of such an initialization. Init strings in the concrete syntax are enclosed in parentheses and separated by a comma. During the compilation of an $\mathcal{SDL}$ expression, this init list then is translated into an Java array of strings. `init()` might throw an `ModuleInitError`.

- `public Object run(Object input) throws ModuleRunError;`
  `run()` starts the execution of the module to which it belongs and returns the result of this computation. As already said, the default implementation of the module methods in class `Modules` (see section 6) does not provide a useful implementation for `run()`—instead an `UnsupportedOperationException` object is thrown. An implementation of `run()` might throw a `ModuleRunError`. It is worth noting 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).

- `public Object setInput(Object input);`
  `setInput()` stores the value of parameter `input` and returns this value.

- `public Object getInput();`
  `getInput()` returns the input originally given to `setInput()`.

- `public Object setOutput(Object output);`
  `setOutput()` stores the value of parameter `output` and returns this value.

- `public Object getOutput();`
  `getOutput()` returns the output originally given to `setOutput()`.

## 5.2 Mediator Interface `IMediator`

We noted already before that every system operator has a corresponding mediator method: + is associated with `seq()`, | with `par()`, and ∗ with `fix()`. A useful default implementation is provided by class `Mediators` (see section 6). We demand that a mediator does not modify the internal state of a module, e.g., it should not use any destructive public methods provided by the module. Especially, a mediator should neither execute `setInput()` nor `setOutput()`. However, it is desirable to read out input and output using `getInput()` and `getOutput()`.

- `public Object seq(IModule module1, IModule module2);`
  `seq()` connects two modules, specified as input parameters `module1` and `module2`. `seq()` returns the proper input for `module2`, computed from the output of `module1`.

- `public Object par(IModule[] modules);`
  `par()` provides a mechanism for combining the output of several modules, indirectly given by the module array `modules`. Conceivable possibilities include the grouping of output results in a new structured object or the computation of a single non-structured value (for instance, an integer number).

- `public Object fix(IModule module);`
  Given a module `module`, `fix()` is intended to iteratively apply the one-argument version of `run()` to the module, until a fixpoint has been reached.

## 5.3 Interface `Runnable`

The option `Threaded = yes` in the concrete syntax of $\mathcal{SDL}$ states that concurrently executed modules have their own computation space and should be realized as Java threads. On a multi-processor system, this option can, in fact, raise the efficiency of the whole system. To inform the Java compiler, such modules must implement the `Runnable` interface, meaning that they must provide an implementation for the zero-argument method `run()`—remember, there already exists a one-argument method of the same name in interface `IModule` (see above).

- `public abstract void run();`
  `run()` starts the execution of the module in a new thread inside the Java virtual machine. Since this nullary version of `run()` neither takes an argument nor does it returns a value, we advise an implementor of the zero-arg `run()` method to memorize the input value via `setInput()` and to store the computed result using `setOutput()`.

# 6 Implementation

We now come to the default implementation of the methods which fulfill the requirements of interface `IModule` and `IMediator`.

## 6.1 Module Methods

The public abstract class `Modules` implements interface `IModule`. Modules might inherit from `Modules` (Java terminology: extends), so that only `run()` must be implemented. Input and output is memorized by introducing the two additional private instance (or member) fields, `input` and `output`.

```
package de.dfki.lt.sdl;

public abstract class Modules implements IModule {

  private Object input;

  private Object output;

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

  public Object run(Object input) throws UnsupportedOperationException {
    throw new UnsupportedOperationException("run(_) is NOT implemented!");
  }

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

  public void init(String[] initArgs) {
    // no effects
  }

  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;
  }

}
```

## 6.2 Mediator Methods

The public class `Mediators` provides a default implementation for the three mediator methods, specified in interface `IMediator`. `seq()` returns the (unmodified) output of `module1`, whereas `par()` groups the output of each module from `modules` in an array. `fix()` returns the fixpoint, but 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 field still contains the original input, whereas the output field refers to the fixpoint, at last.

```
package de.dfki.lt.sdl;

import java.util.*;

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);
  }

}
```

We note here that depending on the structure of `input` and `output` in method `fixpoint`, it is important to have proper implemented `equals()` methods.

## 6.3 *SDL* Compiler

The *SDL* compiler can be activated by calling the public synchronized class method `Sdl.compile()`. The method takes an *SDL* system description, represented as a Java string, and generates a new Java class, whose name is identical to the system name. This Java code can (be edited and) then be compiled by the Java compiler `javac` and the resulting class file can be executed using `java`.

We will now come to the example system from appendix A and utilize it to go into the details of the compilation process. The system has the following form (we omit the Java package names here in the description):

```
System = (| rnd1 rnd2 rnd3) + inc1 + inc2 + (* i5ut42) + (* (| rnd3 rnd2))
Mediator = MaxMediator
Threaded = Yes
rnd1 = Randomize("first", "second", "third")
rnd2 = Randomize("second", "third")
rnd3 = Randomize("third")
inc1 = Increment
inc2 = Increment
i5ut42 = Incr5UpTo42
```

At this point, it is not important to know what the intention of each single module is. First of all, we see that the above expression consists of six variables, some of them reused, viz., `rnd2` and `rnd3`.
These variables are realized in the Java code as private instance fields of class `System` (= name of the new module). The instance fields are prefixed by the dollar sign `$` in order to visually differentiate them from ordinary local variables (marked with the low line character _). The fields are typed to their corresponding classes. These classes are imported, so that they can be directly used without repeating their package prefix. In addition, the above system description requires not to use the standard mediator, instead `MaxMediator` should be employed. Since `System` demands multi-threaded processing (`Threaded = yes`), the class is marked as `Runnable`. When a new instance of `System` is generated, the instance fields are assigned their proper values in the constructor. Some of the modules must also be initialized, viz., `rnd1`, `rnd2`, and `rnd3`.

```
package de.dfki.lt.test;

import java.util.*;
import de.dfki.lt.sdl.*;
import de.dfki.lt.sdl.test.MaxMediator;
import de.dfki.lt.sdl.test.Randomize;
...  // further import statements

public class System extends Modules implements IModule, Runnable {

  private Randomize $rnd1;
  private Randomize $rnd2;
  ...  // further declaration of instance fields

  ...  // a huuuuuge amount of other code

  // the constructor for class System
  public System() throws ModuleInitError {
    super();
    this.$rnd1 = new Randomize();
    this.$rnd2 = new Randomize();
    // further assignments to instance fields (variables)
    ...
    this.$1 = new _Class1();
```

```
      this.$2 = new _Class2();
      // further assignments to instance fields (locally generated classes)
      ...
      String[] _13 = {"first", "second", "third"};
      this.$rnd1.init(_13);
      // further initialization calls
      ...
    }

    // code for the run() method of class System
    public Object run(Object input) throws ModuleClearError, ModuleRunError {
      ...  // code follows later
    }

    // empty run() for Runnable is based on the unary run() directly above
    public void run() throws ModuleClearError, ModuleRunError {
      this.run(this.getInput());
    }

  }
```

The above generated code refers to instance fields not mentioned in the original specification, viz., `$1` and `$2` (plus two additional fields). Since instance fields represent variables in the system description, we might be surprised to which objects `$1` and `$2` refer to. Obviously, they are assigned instances of class `_Class1` and `_Class2`, resp., classes which we have also not introduced (directly).
The solution is quite easy. The above system description is a structured expression, consisting of several subexpressions from which one is even further structured: `(| rnd1 rnd2 rnd3)`, `(* i5ut42)`, `(| rnd3 rnd2)`, and `(* (| rnd3 rnd2))`. The *SDL* compiler automatically introduces so-called *member classes* (Flanagan 2002, pp. 125) for such subexpressions and locates them in the same package to which `System` belongs. Since `$1` and `$2` operate on the same level than the original variables/instance fields, they are also prefixed with `$`. And because new member classes are local to `System`, we consequently prefix them with `_` (as we do for local variables).
The decomposition process always leads to flat expressions of the form `x1 + ... + xk`, `(| x1 ... xk)`, or `(* x)`, resp. `x`, `x1`, ..., `xk` are variables which refer to instances of module classes (see section 3). This decomposition technique is related to what Krieger 1995 calls *normalized type systems*: arbitrary boolean type expressions are decomposed into simpler units which are merely of the form $t_1 \wedge \ldots \wedge t_k$, $t_1 \vee \ldots \vee t_k$, or $\neg t$. $t, t_1, \ldots, t_k$ are either base types or new types which refer to already introduced new type expressions, the analogue to new member classes in our system building approach.
In the above example, we have the following four substitutions (we already use the dollar sign here which the *SDL* compiler has introduced): `$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.
We have now completed the presentation of the very basic idea behind the *SDL* compiler. What comes next is the description of the main `run()` method, the usage of the mediator methods, and the generation of member classes (which

must also implement interface `IModule`). At the end, we also show how to cast the parallel execution of modules in Java code. Appendix A shows several runs of the generated code.

```
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 _14 = this.$1.run(input);
  this.$1.setOutput(_14);
  Object _15 = _med.seq(this.$1, this.$inc1);
  this.$inc1.clear();
  this.$inc1.setInput(_15);
  Object _16 = this.$inc1.run(_15);
  this.$inc1.setOutput(_16);
  Object _17 = _med.seq(this.$inc1, this.$inc2);
  this.$inc2.clear();
  this.$inc2.setInput(_17);
  Object _18 = this.$inc2.run(_17);
  this.$inc2.setOutput(_18);
  Object _19 = _med.seq(this.$inc2, this.$2);
  this.$2.clear();
  this.$2.setInput(_19);
  Object _20 = this.$2.run(_19);
  this.$2.setOutput(_20);
  Object _21 = _med.seq(this.$2, this.$4);
  this.$4.clear();
  this.$4.setInput(_21);
  Object _22 = this.$4.run(_21);
  this.$4.setOutput(_22);
  return this.setOutput(_22);
}
```

At this point, the understanding of the above piece of code for `run()` should not be hard when knowing to what `$1`, ..., `$4` refer to. Note that the sequence mediator `_med.seq()` is called four times due to the fact that + occurs four times in the top-level system specification. It always takes two modules as input (the private `$` fields). The results of the mediator calls (`_15`, `_17`, `_19`, `_21`) serve as input to instances of the individual (sub-)modules (e.g., `this.$inc1.setInput(_15)`). Local variables are also introduced for the return values of the calls to the individual `run()` methods (`_14`, `_16`, `_18`, `_20`, `_22`). These local variables are introduced by the *SDL* compiler to serve as handles (or anchors) to already evaluated subexpression, helping to establish a proper flow of control during the compilation process.
The specification notes `Mediator = de.dfki.lt.sdl.test.MaxMediator` which is reflected by the above code in that the mediator object `_med` is bound to an instance of `MaxMediator`. Notice that the sequence of calls moves from `$1`, to `$inc1`, to `$inc2`, to `$2`, and finally to `$4`, exactly the flow of control as specified in the normalized system description before. There is, however, no `$3` here, since `$3`, which is an instance of `_Class3`, refers to an expression that is used inside `$4`.

When `$4` is executed (`this.$4.run(_21)`), the fixpoint iteration calls `$3` during each iteration step.
Let us now have a look on the generated local class `_Class4` to see how the `run()` method for `$4` works.

```
private class _Class4 implements IModule, Runnable {

  private Object input;

  private Object output;

  private _Class4() {
    this.input = null;
    this.output = null;
  }

  public void clear() {
  }

  ...  // further interface methods from IModule

  public Object run(Object input) throws ModuleClearError, ModuleRunError {
    this.setInput(input);
    IMediator _med = new MaxMediator();
    System.this.$3.clear();
    System.this.$3.setInput(input);
    Object _7 = _med.fix(System.this.$3);
    System.this.$3.setOutput(_7);
    return this.setOutput(_7);
  }

  public void run() throws ModuleClearError, ModuleRunError {
    this.run(this.getInput());
  }

}
```

The above code clearly shows that each local class has to provide code for the methods specified in interface `IModule`. And so it specifies in the class header that it implements `IModule`. Since the original system specification also says that parallel executed modules should be executed as Java threads (`Threaded = Yes`), `_Class4` also makes the nullary `run()` method available. It highlights this in the (local) class header and let us know that it implements `Runnable`.
The `run()` method above is quite easy, since the fixpoint mediator `fix()` encapsulates the code for the fixpoint iteration. The final result is bound to `_7` and to the output field of `$3` (`System.this.$3.setOutput(_7)`) and `$4` (`this.setOutput(_7)`), which is finally returned. The input (and the output) fields of both `$3` and `$4` are not changed during the iteration—the computation of the fixpoint is completely decoupled from the modules (see section 6.2 on how the fixpoint mediator implements this). I.e., even if a fixpoint is reached, the input field still refers to the original input, whereas the output slot memorizes the fixpoint.
Note that we always generate a new mediator object (`_med`) for each local class in order to make the parallel execution of modules thread-safe. The local class

_Class4 refers in the run() method to $3 (the instance which undergoes the iteration). However, $3 is an instance field of the top-level class System. Java provides a construct to cleanly refer to $3: System.this.$3.
We close this section with the presentation of the code for the generated local class _Class3 which concurrently executes $rnd3 and $rnd2 ((| $rnd3 $rnd2)).

```
private class _Class3 implements IModule, Runnable {

  private Object input;

  private Object output;

  private _Class3() {
    this.input = null;
    this.output = null;
  }

  ...  // interface methods from IModule

  public Object run(Object input) throws ModuleClearError, ModuleRunError {
    this.setInput(input);
    IMediator _med = new MaxMediator();
    Thread thread;
    ArrayList threads = new ArrayList();
    System.this.$rnd3.clear();
    System.this.$rnd3.setInput(input);
    thread = new Thread(System.this.$rnd3);
    threads.add(thread);
    thread.start();
    System.this.$rnd2.clear();
    System.this.$rnd2.setInput(input);
    thread = new Thread(System.this.$rnd2);
    threads.add(thread);
    thread.start();
    int pos = -1;
    while (!threads.isEmpty()) {
      for (int i = 0; i < threads.size(); i++) {
        if (!((Thread)(threads.get(i))).isAlive()) {
          pos = i;
          break;
        }
      }
      if (pos != -1) {
        threads.remove(pos);
        pos = -1;
      }
    }
    IModule[] _8 = {System.this.$rnd3, System.this.$rnd2};
    Object _9 = _med.par(_8);
    return this.setOutput(_9);
  }

  public void run() throws ModuleClearError, ModuleRunError {
    this.run(this.getInput());
  }
```

```
}
```

As we see from the code above , the two module instances are initialized first (`clear()`, `setInput(input)`) and then executed as independently running threads (`new Thread(...)`). Within a thread, the zero-argument version of the `run()` method is called. Each thread is added to the local list `threads` which is permanently under inspection. When this list gets empty (i.e., all threads have stopped), the module instances `$rnd3` and `$rnd2` are grouped in an array (`_8`), which is given to the concurrency mediator (`_med.par(_8)`). The result of this mediation then is the final output of calling `$3`.

# 7 Conclusion & Future Work

We have presented the system description language $\mathcal{SDL}$ in which complex new processing systems can be specified by means of a regular expression of already existing modules. At the moment, $\mathcal{SDL}$ provides three operators for expressing a sequence of two modules, a parallel independent execution of arbitrary many modules, and an unrestricted iteration of a single module. Communication between modules is realized by mediators. 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 specification. The execution semantics for the original system specification is defined as the execution of the `run()` method of the generated new Java class.
We have also given a precise formal semantics for syntactically well-formed $\mathcal{SDL}$ expressions which is based on the intuition that modules are functions, whereas meditors correspond to functionals, i.e., functions taking functions as arguments and returning a new function. Composition of modules is defined in terms of mathematical concepts of function theory: Cartesian product, functional composition & application, Lambda abstraction, and unbounded minimization.
Although $\mathcal{SDL}$ has been developed in the context of the shallow language platform *SProUT*, we believe that it can be fruitfully employed in the definition of general software systems. Since the execution semantics is coupled with a precise formal semantics, we think that our approach might be of interest to areas where semantic perspicuity of software systems plays a big role. Related to this is the field of secure software: given a set of secure modules and well-defined combinators, we obtain a well-defined semantics for a new composite module which can perhaps be shown to be secure either. This is clearly future state-of-the-art work.
The approach has been fully implemented in Java, but it should be easy to have a similar system in another language, such as C, C++, or Common Lisp. Moreover, minimal modifications to the $\mathcal{SDL}$ compiler should make it possible not to return Java code, but also programs in other languages which have a notion of either an object, a function, or a function pointer (together with recursion, of course).
There is still room for further improvement. In a complex grown system, it is not unusual that we have modules in different programming languages. The $\mathcal{SDL}$ compiler can only generate a properly running system if it has access to these modules. Since the compiler is programmed in Java, we can make use of the Java Native Interface (JNI; see, e.g., Gordon 1998) and write a specialized Java

class for the external module that implements interface `IModule`. This class would merely consist of native methods which call their corresponding counterparts in the original programming language.
Having only +, |, and $*$ as operators (plus the corresponding mediator methods) might not suffice at all. By adding more and more such concepts to $\mathcal{SDL}$, we will end in a generalized programming language which has modules and operators as building blocks. We can envisage additional useful constructs:

- *assignment*
  Non-linear processing systems might find it interesting to have a built-in mechanism for storing information obtained so far, viz., variables:
  ```
  m1 + var:=m2 + m3 + (| var m4)
  ```
  Read this example as if `var:=m2` is put in parentheses, i.e., `:=` has a higher priority than `+`. `var` here serves as a constant function (return value equals to the output of `m2`) which is executed in parallel to `m5`.

- *if-then-else*
  It is conceivable that depending on the form of the output of a certain module `m`, either module `m1` or module `m2` should be executed:
  ```
  ...  var:=m ...  if (pred(var)) then m1 else m2
  ```
  In the above example, it is important to have both variables (`var`) and predicates (`pred(...)`) which return Boolean values. A conditional, such as `if` can, of course, be realized as a variant of the | operator, together with a specialized mediator, thus being only syntactic sugar in most cases.

- *distributor*
  It is not unlikely that results produced by several modules `m1`, ..., `mj` need to be rated according to a given measure and distributed to succeeding modules `n1`, ..., `nk` ($j \geq k$), whereas the best-ranked result is given to the first module, the second-best to the second module, and so on. The syntax could be
  ```
  ...  (| m1 ...  mj) + (< n1 ...  nk) + ...
  ```
  and $<$ indicates that `n1`, ..., `nk` get the ranked results from `m1`, ..., `mj`. (Thanks Anette for the example.)

- *first come wins*
  Instead of executing modules in parallel under `|` and collecting their results in a structured object, we might relocate their execution in a specialized mediator `!` and let the fastest module win:
  ```
  ...  (!  m1 ...  mj) + n ...
  ```
  (Thanks Feiyu.)

Contrary to dynamic object-oriented programming languages such as CLOS (Keene 1989), Java enforce classes which implements a certain interface to have methods that exactly match the corresponding header in the interface. I.e., if a module is guaranteed to read and return only, say, `Integer` objects, we still have to use (and perhaps implement) the interface methods `getInput()`, `setInput()`, `getOutput()`, `setOutput()`, and `run()` which operate on the most general class `Object`.

Thus, the Java compiler can not statically check at compile time if a system description is valid in terms of the domain and range of each individual module. Assume, for instance, that *square* reads and returns floats and that *duplicate* maps from strings onto strings. In the current system, we can neither determine that *square* ∘ *duplicate* nor *duplicate* ∘ *square* is an invalid function, since the interface methods are typed to `Object` which will always result in a proper description. Due to this fact, it seems adequate to implement a proper domain/range check. This implementation, however, has to make use of Java's reflection API, since information concerning the sub-/supertype relationship between classes/interfaces must be obtained. This check is assumed to be part of the next major release of *SDL*.

# A Example

This appendix presents several runs of the example system which we have already studied in the previous sections. Its definition is

```
System = (| rnd1 rnd2 rnd3) + inc1 + inc2 + (* i5ut42) + (* (| rnd3 rnd2))
Mediator = MaxMediator
Threaded = Yes
rnd1 = Randomize("first", "second", "third")
rnd2 = Randomize("second", "third")
rnd3 = Randomize("third")
inc1 = Increment
inc2 = Increment
i5ut42 = Incr5UpTo42
```

The intention behind the individual modules is as follows:

- `Randomize` returns natural random numbers between 0 and $n - 1$, where $n$ is the input given to the randomizer; in case that the input is zero, the output is defined to be zero; the init strings given to `Randomize` are printed to `System.out` one after another;
- `Increment` increments the input (a number) by one;
- `Incr5UpTo42` increments the input by five for numbers less than 42; an input greater or equal than 42 is always mapped onto 42.

Let us have a look on the definition of the last module:

```
public class Incr5UpTo42 extends Modules {

  public Incr5UpTo42() {
    super();
  }

  public Object run(Object obj) {
    if (obj instanceof Integer) {
      int i = ((Integer)obj).intValue();
      if (i < 42) {
```

```
        System.out.println(i + " --> Incr5UpTo42 --> " + (i + 5));
        return new Integer(i + 5);
      }
      else {
        System.out.println(i + " --> Incr5UpTo42 --> 42");
        return new Integer(42);
      }
    }
    else
      throw new ModuleRunError("Module Incr5UpTo42 is given a wrong class");
  }

}
```

Since the interface methods are typed to `Object` and since integer numbers are only a primitive data type in Java, we employ the corresponding wrapper class `Integer` here. The default mediator object is replaced in the above definition by `MaxMediator`, which computes the maximum number from the output of an arbitrary number of modules, instead of grouping the output in an array:

```
public class MaxMediator extends Mediators {

  public MaxMediator() {
    super();
  }

  public Object par(IModule[] modules) {
    int[] outs = new int[modules.length];
    int i;
    for (i = 0; i < modules.length; i++)
      outs[i] = ((Integer)(modules[i].getOutput())).intValue();
    int max = outs[0];
    for (i = 0; i < outs.length; i++)
      if (outs[i] > max)
        max = outs[i];
    System.out.print("<");
    for (i = 0; i < outs.length; i++){
      System.out.print(outs[i]);
      if ((i + 1) != outs.length)
        System.out.print(", ");
    }
    System.out.print(">");
    System.out.println(" --> MaxMediator --> " + max);
    return new Integer(max);
  }

}
```

Let us now finish this paper and have a look on two successive runs of the Java code of the above system. The output of both runs clearly differ due to the use of the randomizers.

```
0 krieger@leitwort (~/Java/java) 576 $ java de/dfki/lt/test/System 10
third
second
```

```
first
10 --> Randomize --> 1
10 --> Randomize --> 4
10 --> Randomize --> 3
<1, 4, 3> --> MaxMediator --> 4
4 --> Increment --> 5
5 --> Increment --> 6
6 --> Incr5UpTo42 --> 11
11 --> Incr5UpTo42 --> 16
16 --> Incr5UpTo42 --> 21
21 --> Incr5UpTo42 --> 26
26 --> Incr5UpTo42 --> 31
31 --> Incr5UpTo42 --> 36
36 --> Incr5UpTo42 --> 41
41 --> Incr5UpTo42 --> 46
46 --> Incr5UpTo42 --> 42
42 --> Incr5UpTo42 --> 42
42 --> Randomize --> 28
42 --> Randomize --> 5
<28, 5> --> MaxMediator --> 28
28 --> Randomize --> 2
28 --> Randomize --> 6
<2, 6> --> MaxMediator --> 6
6 --> Randomize --> 1
6 --> Randomize --> 3
<3, 1> --> MaxMediator --> 3
3 --> Randomize --> 1
3 --> Randomize --> 2
<1, 2> --> MaxMediator --> 2
2 --> Randomize --> 1
2 --> Randomize --> 0
<1, 0> --> MaxMediator --> 1
1 --> Randomize --> 0
1 --> Randomize --> 0
<0, 0> --> MaxMediator --> 0
0 --> Randomize --> 0
0 --> Randomize --> 0
<0, 0> --> MaxMediator --> 0
```

And now the next run.

```
0 krieger@leitwort (~/Java/java) 577 $ java de/dfki/lt/test/System 10
third
second
first
10 --> Randomize --> 3
10 --> Randomize --> 6
10 --> Randomize --> 5
<3, 6, 5> --> MaxMediator --> 6
6 --> Increment --> 7
7 --> Increment --> 8
8 --> Incr5UpTo42 --> 13
13 --> Incr5UpTo42 --> 18
18 --> Incr5UpTo42 --> 23
23 --> Incr5UpTo42 --> 28
28 --> Incr5UpTo42 --> 33
33 --> Incr5UpTo42 --> 38
```

```
38 --> Incr5UpTo42 --> 43
43 --> Incr5UpTo42 --> 42
42 --> Incr5UpTo42 --> 42
42 --> Randomize --> 29
42 --> Randomize --> 5
<29, 5> --> MaxMediator --> 29
29 --> Randomize --> 1
29 --> Randomize --> 2
<1, 2> --> MaxMediator --> 2
2 --> Randomize --> 0
2 --> Randomize --> 1
<0, 1> --> MaxMediator --> 1
1 --> Randomize --> 0
1 --> Randomize --> 0
<0, 0> --> MaxMediator --> 0
0 --> Randomize --> 0
0 --> Randomize --> 0
<0, 0> --> MaxMediator --> 0
```

# References


Abney, S. 1996. Partial Parsing via Finite-State Cascades. *Natural Language Engineering* 2(4):337–344.

Arnold, K., J. Gosling, and D. Holmes. 2000. *The Java Programming Language*. Boston: Addison-Wesley. 3rd edition.

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

Becker, M., W. Drożdżyński, H.-U. Krieger, J. Piskorski, U. Schäfer, and F. Xu. 2002. SProUT—Shallow Processing with Unification and Typed Feature Structures. In *Proceeding of the International Conference on Natural Language Processing, ICON-2002*.

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

Carpenter, B. 1992. *The Logic of Typed Feature Structures*. Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press.

Crysmann, B., 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 the 40th Annual Meeting of the Association for Computational Linguistics, ACL-2002*, 441–448.

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

Flanagan, D. 2002. *Java in a Nutshell*. Beijing: O'Reilly. 4th edition.

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

Gordon, R. 1998. *Essential JNI: Java Native Interface.* Upper Saddle River, NJ: Prentice Hall.

Hermes, H. 1978. *Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit: Einführung in die Theorie der rekursiven Funktionen.* Vol. 87 of Heidelberger Taschenbücher. Berlin: Springer. 3rd edition. In German. Also in English as *Enumerability, Decidability, Computability: An Introduction to the Theory of Recursive Functions.*

Hobbs, J., D. Appelt, J. Bear, D. Israel, M. Kameyama, M. Stickel, and M. Tyson. 1997. FASTUS: A Cascaded Finite-State Transducer for Extracting Information from Natural-Language Text. In *Finite State Devices for Natural Language Processing*, ed. E. Roche and Y. Schabes. MIT Press.

Keene, S. E. 1989. *Object-Oriented Programming in* Common Lisp. *A Programmer's Guide to* CLOS. Reading, MA: Addison-Wesley.

Krieger, H.-U. 1995. *TDL—A Type Description Language for Constraint-Based Grammars. Foundations, Implementation, and Applications.* PhD thesis, Universität des Saarlandes, Department of Computer Science, September.

# $\mathcal{SDL}$—A System Description Language

**Hans-Ulrich Krieger**

**RR-03-01**
Research Report