Non-intrusive formal methods and strategic rewriting for a chemical application

Oana Andrei, Liliana Ibanescu, Hélène Kirchner

Non-intrusive formal methods and strategic rewriting for a chemical application

Oana Andrei[1], Liliana Ibanescu[2], and Hélène Kirchner[3]

[1] INRIA - LORIA
[2] ULP Strasbourg
[3] CNRS - LORIA
Campus scientifique BP 239
F-54506 Vandoeuvre-lès-Nancy Cedex, France
First.Last@loria.fr

Abstract. The concept of formal islands allows adding to existing programming languages, formal features that can be compiled later on into the host language itself, therefore inducing no dependency on the formal language. We illustrate this approach with the TOM system that provides matching, normalization and strategic rewriting, and we give a formal island implementation for the simulation of a chemical reactor.

1 Introduction

Concerned by the crucial need for improvement of existing software in their logic, algorithmic, security and maintenance qualities, formal methods are more and more used in the software design process. Usually they come into play both at the design and verification levels either for formal specification or high-level programming. But this approach does not take into account existing software, while thousands of code lines are executed every day. This might be one of the reasons why formal methods did not yet fully succeed at the industrial level.

Among many formal method approaches, algebraic techniques providing a clear semantics for signatures and rewrite rules are used in high-level languages and environments like ASF+SDF [24], Maude [8], Cafe-OBJ [10], or ELAN [5, 17] which have been designed on these concepts. These rule-based systems have gained considerable interest with the development of efficient compilers. However, when programs are developed in these languages, they can hardly interact with programs written in another language like C or Java.

The work presented here proposes an alternative reconciling the use of algebraic formal features with the widely used object-oriented language Java. This is possible through the *Formal Islands* approach developed in the Protheo project team since a few years [20]. A formal island is a piece of code introducing formal features. These new features are anchored in terms of the available functionalities of the host language. Once compiled, these features are translated into pure host language constructs. The formal island concept is implemented through

the software system TOM [2] that builds upon the concepts of rules and strategic rewriting. TOM is already used for transforming XML documents, and the compiler itself has been designed with TOM.

The approach and the use of TOM are illustrated in this paper with a specific example: we apply strategic rewriting to model a chemical reactor by means of a formal island implementation. The considered problem is the automated generation of reaction mechanisms: a set of molecules and a list of generic elementary reactions (reaction patterns) are given as input to a generator that produces the list of all possible elementary reactions according to a specific reactor dynamics.

A number of software systems [4, 13, 21, 29, 31] have been developed for the automated generation of reaction mechanisms [9, 23]. As far as literature says, these systems are implemented using traditional programming languages, employing rather ad-hoc data structures and procedures for the representation and transformations of molecules (e.g. Boolean adjacency matrices and matrices transformations). Furthermore, existing systems are limited, sometimes by their implementation technology, to acyclic species, or mono-cyclic species, whereas combustion mechanisms often involve aromatic species, which are polycyclic.

In GasEl project [6, 7, 15] we already have explored the use of rule-based systems and strategies for the problem of automated generation of kinetics mechanisms [23, 9] in the whole context of its use by chemists and industrial partners. In GasEl the representation of chemical species uses the notion of molecular graphs, encoded by a term structure called GasEl terms [6] which is inspired by the linear notation SMILES [30]. The graph isomorphism test is based on the Unique SMILES algorithm [30]. Reactions patterns are encoded by a set of conditional rewriting rules on GasEl terms. The molecular graph rewriting relation is simulated by a rewriting relation on equivalence classes of terms [7]. The control of the chemical reactions chaining (i.e. reactor dynamics) is described using a strategy language [6]. GasEl prototype is implemented in ELAN [5, 17], encoding a set of nine reaction patterns. Qualitative validations have been performed with chemists [15].

The formal background of strategic rewriting is quite relevant for the considered problem: (i) chemical reactions are naturally expressed by chemists themselves using conditional rules; (ii) matching power associated with rewriting allows retrieving patterns in chemical species, (iii) defining the control on rules is essential for designing automated mechanisms generators in a flexible way and controlling combinatorial explosion. The main technical difficulty in ELAN implementation consisted in the encoding of reaction patterns on GasEl terms that correctly simulates the corresponding transformation on molecular graphs. The TOM implementation provides another approach to this problem, while keeping the same molecular graph rewriting relation, and preserving the same chemical principles and hypotheses as in GasEl.

The paper is structured as follows. Section 2 presents the formal island concept that will be further illustrated in the sequel. The TOM system is briefly described in Section 3 and the main language constructions needed to understand the considered application are introduced. Section 4 is devoted to the chemical

example and explains what kind of reactor is modelled. Section 5 addresses the formal island implementation of the chemical reactor and details the different steps performed to achieve the Java implementation. Finally Section 6 draws some conclusions and perspectives for future work.

2 Formal islands

Since several years, we have been strongly concerned with the feasibility of strategic rewriting as a practical programming paradigm [1, 17]. The development of efficient compilation concepts and techniques took an important place in the language support design. The results presented in [19] led to a quite efficient implementation and thus demonstrated the practicality of the paradigm.

Making strategic rewriting easily available in many programming languages was the main concern that led to the emergence of *formal island*. This concept provides a general way to make formal methods, and in particular matching and rewriting widely available.

We use the notions of *formal island* and *anchoring* to extend an existing language with formal capabilities. A formal island is a piece of code introducing formal features, while anchoring means to describe these new features in terms of the available functionalities of the host language. Once compiled, these features are translated into pure host language constructs.

We review the definitions of representation and formal anchor for the unsorted case, and a small example of formal anchoring from [18].

In order to precisely define these notions, we recall a few concepts of first order term algebra needed here [16]. A *signature* $\mathcal{F}$ is a set of function symbols, each one associated to a natural number by the arity function, $ar : \mathcal{F} \to \mathbb{N}$. $\mathcal{F}_n$ is the set of function symbols of arity n, $\mathcal{F}_n = \{f \in \mathcal{F} \mid ar(f) = n\}$. $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the set of *terms* from a given finite set $\mathcal{F}$ of function symbols and a denumerable set $\mathcal{X}$ of variable symbols. A *position* within a term is represented as sequence ω of positive integers describing the path from the root of the term to the root of the subterm at that position, denoted by $t_{|\omega}$. $Symb(t)$ is a partial function from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to $\mathcal{F}$ which associates to each term t its root symbol $f \in \mathcal{F}$. The set of variables occurring in a term t is denoted by $Var(t)$. If $Var(t)$ is empty, t is called a *ground term* and $\mathcal{T}(\mathcal{F})$ is the set of ground terms. Two ground terms t and u of $\mathcal{T}(\mathcal{F})$ are equals, and we denote this by $t = u$, when, for some function symbol f, $Symb(t) = Symb(u) = f$, $f \in \mathcal{F}_n$, $t = f(t_1, \ldots, t_n)$, $u = f(u_1, \ldots, u_n)$, and $\forall i \in [1..n]$, $t_i = u_i$.

Definition 1. ([18]) *Given a tuple composed of a signature $\mathcal{F}$, a set of variables $\mathcal{X}$, booleans $\mathbb{B}$ and integers $\mathbb{N}$, given sets of host language constructs $\Omega_{\mathcal{F}}$, $\Omega_{\mathcal{X}}$, $\Omega_{\mathcal{T}}$, $\Omega_{\mathbb{B}}$, and $\Omega_{\mathbb{N}}$, we consider a family of* representation *functions $\ulcorner\urcorner$ that map:*

- *function symbols $f \in \mathcal{F}$ to elements of $\Omega_{\mathcal{F}}$, denoted $\ulcorner f \urcorner$,*
- *variables $v \in \mathcal{X}$ to elements of $\Omega_{\mathcal{X}}$, denoted $\ulcorner v \urcorner$,*
- *ground terms $t \in \mathcal{T}(\mathcal{F})$ to elements of $\Omega_{\mathcal{T}}$, denoted $\ulcorner t \urcorner$,*
- *booleans $b \in \mathbb{B} = \{\top, \bot\}$ to elements of $\Omega_{\mathbb{B}}$, denoted $\ulcorner b \urcorner$,*

– *natural numbers* $n \in \mathbb{N}$ *to elements of* $\Omega_{\mathbb{N}}$*, denoted* $\ulcorner n \urcorner$*.*

Definition 2. ([18]) *Given a tuple* $\langle \mathcal{F}, \mathcal{X}, \mathcal{T}(\mathcal{F})\mathbb{B}, \mathbb{N} \rangle$*, and the operations* eq: $\Omega_{\mathcal{T}} \times \Omega_{\mathcal{T}} \rightarrow \Omega_{\mathbb{B}}$*,* is_fsym: $\Omega_{\mathcal{T}} \times \Omega_{\mathcal{F}} \rightarrow \Omega_{\mathbb{B}}$*, and* $\mathsf{subterm}_f$*:* $\Omega_{\mathcal{T}} \times \Omega_{\mathbb{N}} \rightarrow \Omega_{\mathcal{T}}$ $(f \in \mathcal{F})$*, a representation function* $\ulcorner \urcorner$ *is a* formal anchor *if it preserves the structural properties of* $\mathcal{T}(\mathcal{F})$ *in* $\ulcorner \mathcal{T}(\mathcal{F}) \urcorner$ *by the semantics of* eq*,* is_fsym*, and* $\mathsf{subterm}_f$*:*
$\forall\ t, t_1, t_2 \in \mathcal{T}(\mathcal{F}), \forall f \in \mathcal{F}, \forall i \in [1..ar(f)]\ :$

$$\mathsf{eq}(\ulcorner t_1 \urcorner, \ulcorner t_2 \urcorner) \equiv \ulcorner t_1 = t_2 \urcorner$$
$$\mathsf{is\_fsym}(\ulcorner t \urcorner, \ulcorner f \urcorner) \equiv \ulcorner Symb(t) = f \urcorner$$
$$\mathsf{subterm}_f(\ulcorner t \urcorner, \ulcorner i \urcorner) \equiv \ulcorner t_{|i} \urcorner \text{ if } Symb(t) = f$$

We illustrate the concept of formal anchor with a small example from [18]:

Example 1. In C or Java like languages, the notation of *term* can be implemented by a record *(sym:integer, sub:array of term)*, where the first slot *(sym)* denotes the top symbol, and the second slot *(sub)* corresponds to the subterms. It is easy to check that the following definitions of eq, is_fsym, and $\mathsf{subterm}_f$ (where $=$ denotes an atomic equality) provide a formal anchor for $\mathcal{T}(\mathcal{F})$:

$$\begin{aligned} \mathsf{eq}(t_1, t_2) &\triangleq t_1.sym = t_2.sym \ \wedge\ \forall i \in [1..ar(t_1.sym)], \\ &\quad \mathsf{eq}(t_1.sub[i], t_2.sub[i]) \\ \mathsf{is\_fsym}(t, f) &\triangleq t.sym = f \\ \mathsf{subterm}_f(t, i) &\triangleq t.sub[i] \text{ if } t.sym = f \wedge i \in [1..ar(f)] \end{aligned}$$

3 TOM

TOM is an implementation of the idea of formal island [20]. TOM [2] provides matching, normalization, and strategic rewriting in Java, C, and Caml [20, 14]. In particular, we have used Java for developing the chemical application described in this paper. In each of the three instances, matching and rewriting primitives can be combined with constructs of the programming language, then compiled to the host language, using similar techniques as for compiling ELAN. The normal forms provided by rewriting are available to get conciseness and expressiveness in programs written in the host language. Moreover one can prove that these sets of rewrite rules have useful properties like termination or confluence. Once the programmer has used rewriting to specify functionalities and to prove properties, the compilation dissolves this formal island in the existing code. The use of rewriting and TOM therefore induces no dependence: once compiled, a TOM program contains no more trace of the rewriting and matching statements that were used to build it.

Basically, a TOM program is a list of blocks, where each block is either a TOM construct, or a sequence of characters. The idea is that after transformation, the sequence of characters merged with the compiled TOM constructs becomes a valid host language program.

The main construct, `%match`, is similar to the match primitive found in functional languages: given an object (called subject) and a list of patterns-actions,

the match primitive selects the first pattern that matches the subject and performs the associated action. The subject against which we match can be any object, but in practice, this object is usually a tree-based data-structure, also called term in the algebraic programming community. The match construct may be seen as an extension of the classical switch/case construct. The main difference is that the discrimination occurs on a term and not on atomic values like characters or integers: the patterns are used to discriminate and retrieve information from an algebraic data structure.

In addition to `%match` TOM provides the `%rule` construct which allows describing rewrite rule systems. This construct supports conditional rewrite rules as well as rules with matching conditions (as in ELAN or ASF+SDF). By default, TOM rules provide a leftmost innermost normalization strategy which computes normal forms in an efficient way. It is of course possible to combine these features with more complex strategies, like generic traversal strategies, to describe more complex or generic transformations. When understanding all the possibilities offered by TOM, this general purpose system becomes as powerful and expressive as many specific rewrite rule based programming languages.

Another construct of TOM is the backquote (‘). This construct is used for building an algebraic term or to retrieve the value of a TOM variable (a variable instantiated by pattern-matching).

$Identity@(t)$	$=> t$
$Fail@(t)$	$=> failure$
$Sequence(s_1, s_2)@(t)$	$=> failure$ if $s_1@(t)$ fails
	$s_2@(t')$ if $s_1@(t) => t'$
$Choice(s_1, s_2)@(t)$	$=> t'$ if $s_1@(t) => t'$
	$s_2@(t)$ if $s_1@(t)$ fails
$All(s)@(f(t_1, ..., t_n))$	$=> f(t'_1, ..., t'_n)$ if $s_1@(t_1) => t'_1, ..., s_n@(t_n) => t'_n$
	$failure$ if there exists i such that $s_i@(t_i)$ fails
$All(s)@(cst)$	$=> cst$
$One(s)@(f(t_1, ..., t_n))$	$=> f(t_1, ..., t'_i, ..., t_n)$ if $s_i@(t_i) => t'_i$
	$failure$ if $s_1@(t_1)$ fails, ..., $s_n@(t_n)$ fails
$One(s)@(cst)$	$=> failure$
$Omega(i, s)@(f(t_1, ..., t_n))$	$=> f(t_1, ..., t'_i, ..., t_n)$ if $s@(t_i) => t'_i$
	$failure$ if $s@(t_i)$ fails

Fig. 1. Strategy constructors

The `%vas` construct allows the user to define a many-sorted signature. This construct is replaced at compilation time by the content of the generated formal anchor.

Other available constructs like `%typeterm`, `%typelist`, and `%op` which define formal anchor between signature formalism and concrete implementations (Java classes) allow performing pattern matching against any data structure.

In order to make easier the use of TOM, two tools were developed: ApiGen and Vas [2]. ApiGen is a system which takes a many-sorted signature as input, and it generates both a concrete implementation for the abstract data-type (for

example Java classes), and a mapping for TOM. Vas is a preprocessor for ApiGen which provides a human-readable syntax definition formalism inspired from SDF. These two systems are useful for manipulating Abstract Syntax Trees since they offer an efficient implementation based on ATerms [25] which supports maximal memory sharing, strong static typing, as well as parsers and pretty-printers.

TOM provides a library inspired by ELAN, Stratego [26], and JJTraveler [28], which allows us to easily define various kind of traversal strategies. Figure 1 provides an algebraic view of elementary strategy constructors, and defines their evaluation using the application operator @. In this framework, the application of a strategy to a term can fail. In Java, the failure is implemented by an exception (`VisitFailure`).

These strategy constructors are the key-component that can be used to define more complex strategies. In order to define recursive strategies, the μ abstractor was introduced. This allows giving a name to the current strategy, which can be referenced later. Using strategy operators and the μ abstractor, new strategies can be defined [27] as illustrated by Figure 2.

$$
\begin{array}{ll}
Try(s) & = Choice(s, Identity) \\
Repeat(s) & = \mu x.Choice(Sequence(s, x), Identity()) \\
BottomUp(s) & = \mu x.Sequence(All(x), s)) \\
TopDown(s) & = \mu x.Sequence(s, All(x))) \\
Innermost(s) & = \mu x.Sequence(All(x), Try(Sequence(s, x)))
\end{array}
$$

Fig. 2. Examples of strategies

The Try strategy never fails: it tries to apply the strategy s. If it succeeds, the result is returned. Otherwise, the $Identity$ strategy is applied, and the subject is not modified.

The *Repeat* strategy applies the strategy s as many times as possible, until a failure occurs. The last unfailing result is returned.

The strategy $BottomUp$ tries to apply the strategy s to all nodes, starting from the leaves. Note that the application of s should not fail, otherwise the whole strategy also fails.

The $TopDown$ strategy tries to apply the strategy s to all nodes, starting from the root. It fails if the application of s fails at least once.

The strategy $Innermost$ tries to apply s as many times as possible, starting from the leaves. This construct is useful to compute normal forms.

4 Strategic rewriting for a chemical reactor

The purpose of an automated generator of detailed kinetic mechanisms is to take as input one or more hydrocarbon molecules and the reaction conditions, and to give as output a reaction model, i.e. the list of applied reactions.

In this section we present the model used for the representation of chemical species, the reaction pattern we considered, and the reactor dynamics.

4.1 Molecular graphs

We now describe more formally the chemical model we want to implement.

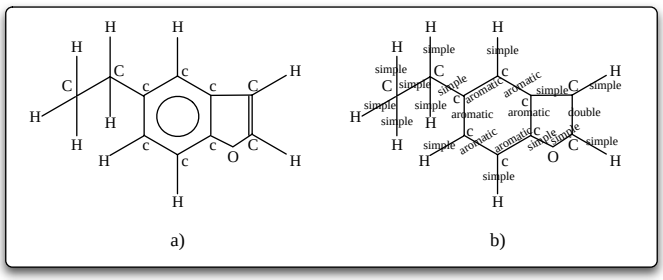

Fig. 3. Molecular graphs

A molecular graph [11] is a vertex-labelled and edge-labelled graph, where each vertex is labelled with an atom and each edge is labelled with the bond type, as illustrated in Figure 3. A chemical reaction is expressed as a rewriting rule for molecular graphs. Figure 4 gives an example of a chemical reaction.

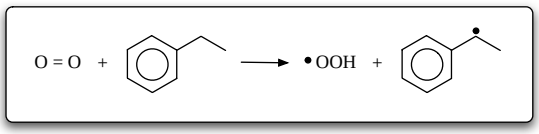

Fig. 4. Bimolecular initiation for ethylbenzene

4.2 Rules for decorated labelled graphs

In the so-called primary mechanism, a set of nine reaction patterns is applied to an initial mixture of molecules. A complete description of the involved reactions patterns is out of the scope of this paper, but the chemistry-like presentation from Figure 5 gives the flavor of the transformations needed to be encoded.

Every reaction pattern is actually also guarded by "chemical filters", i.e. chemical conditions of applications, not mentioned here, even if several of them are currently implemented: they include considerations on the number of atoms in involved molecules or free radicals, the type of radicals or the type of bonds, etc. Some of them are discussed in [9].

In the GasEl prototype, small molecules (like $\mathbf{O} = \mathbf{O}, \bullet\mathbf{H}$, β radicals ([15]), $\bullet\mathbf{O}\bullet$, etc.) can always be injected in the set of chemical reactants or we assume they exist in order to allow reactions like (bi), (me), (ipso), (ox), (co.O.) to be applied. In the current implementation we always work with an explicit population of reactants.

Name	Description
ui	$x - y \longrightarrow x\bullet + \bullet y$
bi	$\mathbf{O=O} + \mathbf{H} - x \longrightarrow \bullet\mathbf{OOH} + \bullet x$
ipso	$\bullet\mathbf{H} + Ar - x \longrightarrow \mathbf{H} - Ar + \bullet x$
me	$\bullet\beta + \mathbf{H} - x \longrightarrow \beta - \mathbf{H} + \bullet x$
bs	$\bullet x - y - z \longrightarrow x = y + \bullet z$
ox	$\mathbf{O=O} + \mathbf{H} - x - y\bullet \longrightarrow \bullet\mathbf{OOH} + x = y$
co.O.	$\bullet\mathbf{O}\bullet + \bullet x \longrightarrow \bullet\mathbf{O} - x$
co	$\bullet x + \bullet y \longrightarrow x - y$
di	$\bullet x + \mathbf{H} - y - z\bullet \longrightarrow x - \mathbf{H} + y = z$

Fig. 5. Reaction patterns of primary mechanism: patterns involve simple ($-$) or double ($=$) bonds, free radicals ($\bullet x$), specific atoms ($\mathbf{O}$, $\mathbf{H}$); variables x, y, z can be instantiated by any reactants

4.3 Primary mechanism

The primary mechanism can be described as the result of three stages (see Figure 6):

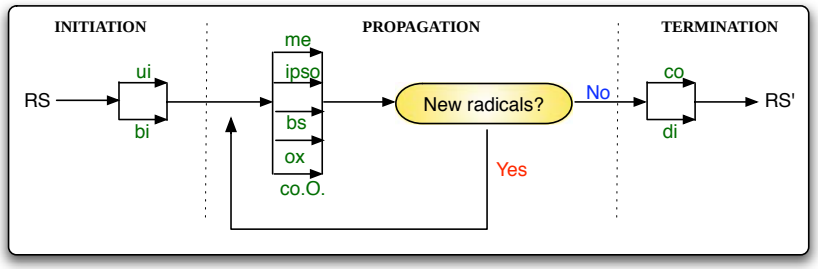

Fig. 6. Primary mechanism

1. The *initiation* stage: unimolecular and bimolecular initiation reactions, (ui) and (bi), are applied to initial reactants, i.e. to the initial mixture of molecules. Let RS_1 = RS be the set of all reactants that can be obtained.
2. The *propagation* stage: a set of generic patterns of reactions, (ipso), (me), (bs), (ox), and (co.O.), are applied to all reactants in RS_i to obtain a new set RS_{i+1} of reactants. RS_{i+1} consists in all reactants of RS_i plus those that can be obtained by these reactions. This is iterated until no new reactant is generated.
3. The *termination* stage: combination and disproportionation reactions, (co) and (di), are applied to free radicals of RS_i to get a set `RS'` of molecules.

The set of reaction rules R is partitioned in three sets R_i, R_p, and R_t where $R_i = \{$(ui), (bi)$\}$, $R_p = \{$(me), (ipso), (bs), (ox), (co.O.)$\}$, and $R_t = \{$(co), (di)$\}$.

For expository reasons we consider that all reactions have the generic form $m_1 + m_2 \rightarrow m_1' + m_2'$, where at most one reactant in each side of the rule can be a "dummy" reactant which is always present in a set of reactants.

```
P(S) UNIT (R : P(R), P_1 : P(S) [, P_2 : P(S)])
  begin
    P' := ∅;
    while(¬terminate()) do
      (m_1, m_2) := select(P_1 [, P_2]);
      for all (m_1 + m_2 → m'_1 + m'_2) ∈ R
          P' := insert(P', m'_1, m'_2);
      fi
    od
    return P'
  end
```

Fig. 7. The UNIT algorithm

The algorithms for the reactor behavior in each stage have a common part, which we call UNIT (Figure 7), parametrized by a set of reaction rules and one or two input sets of reactants. $select(P_1)$ returns a randomly chosen pair of reactants from P_1 not chosen before, without removing them from P_1. $select(P_1, P_2)$ returns also a randomly chosen pair of reactants, first from P_1 and the second from P_2, not chosen before, without removing the reactants from the two sets. $insert(P, m'_1, m'_2)$ adds the two reaction products m'_1 and m'_2 to P if they are not already in P. The function $terminate()$ returns $false$ as long as there are reactants that can interact by means of rules from R.

```
P(S) AlgInit (P_0 : P(S))
  begin
    return P_0 ∪ UNIT(R_i, P_0)
  end

P(S) AlgTermin (P_0 : P(S))
  begin
    return P_0 ∪ UNIT(R_t, P_0)
  end

P(S) AlgPropag (P_0 : P(S))
  begin
    i := 0;
    P'' := ∅;
    repeat
      P'' := P'' ∪ P_i;
      P_{i+1} := UNIT(R_p, P'', P_i) \ P'';
      i := i + 1;
    until P_i = ∅;
    return P'';
  end
```

Fig. 8. The stage algorithms

Now the algorithms for the three stages can be written in a rather uniform presentation as given in Figure 8.

We consider that the three stages of the reactor are executed sequentially due to chemical hypothesis. Therefore the reactor dynamics is described by Figure 9.

$$(AlgInit; AlgPropag; AlgTermin)(P_0)$$

Fig. 9. The reactor dynamics

5 A formal island implementation of the primary mechanism

Let us now consider in this section how the primary mechanism is implemented in TOM using the formal island principle.

The TOM implementation involves four steps, in order to design:

1. An algebraic view of molecular graphs, as a set of terms on a convenient signature.
2. A representation mapping that establishes a correspondence between algebraic terms and Java objects. (This is the formal anchor.)
3. Reaction rules implemented with match constructs: the left-hand side consists in a TOM term, while the right-hand side is a mixture of Java code and TOM constructs.
4. Strategies for applying the reaction rules within each stage, and the chaining of stages.

In the following subsections we develop each of these steps.

5.1 Molecular graphs viewed as algebraic terms

A molecular graph (see Figure 3) is encoded by a term, as proposed in the linear notation SMILES presented in [30]. We briefly recall the principles of this representation. Molecules are represented as hydrogen-suppressed molecular graphs (hydrogen atoms are not represented) with atom-labelled vertices and bond-labelled edges. If the hydrogen-suppressed molecular graph has cycles, it can be transformed into a tree applying the following rule to every cycle: choose

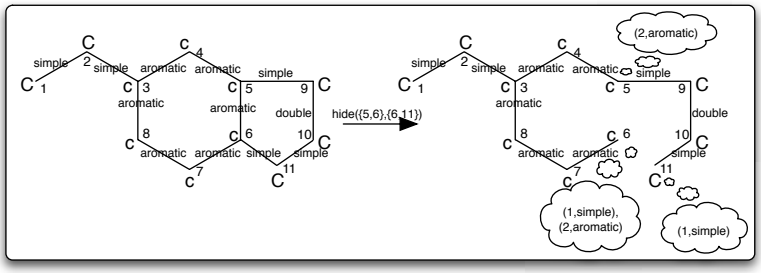

Fig. 10. From a cyclic molecular graph to an acyclic decorated molecular graph

one fresh digit and one simple- or aromatic-labelled edge of the cycle, delete the edge, and add the same digit and the label of the edge to the labels of the formerly adjacent vertices. A vertex is chosen as root, and the tree is represented in a "semi-well-formed" parenthesized preorder traversal (the parentheses are omitted for the right-most child of each vertex). Moreover, an aromatic cycle is represented by lower case letters, the aromatic and simple bonds are not represented.

In the first molecular graph from Figure 10 two edges are transformed into *implicit* edges: (i) edge {6,11} labelled with *simple* is hidden and encoded by labels $(1, simple)$ on vertices 6 and 11; (ii) edge {5,6} labelled with *aromatic* is hidden and encoded by labels $(2, aromatic)$ on vertices 5 and 6. The aromaticity of a bond is propagated to its end vertices which are labelled by lower case letters in the SMILES notation, and by lower case letters prefixed by `ar` in the signature. For example, if the vertex number 1 is chosen as root, a linear notation is `CCc(ccc12)cc2C=CO1`; if the root is the vertex number 3 another notation is `C(CC)(ccc12)cc2C=CO1`.

```
sorts Atom Bond CLabel CLabelList Symbol Reactant ReactantList
abstract syntax
  C   -> Atom
  arC -> Atom
  O   -> Atom
  H   -> Atom
  e   -> Atom
  none   -> Bond
  simple -> Bond
  double -> Bond
  triple -> Bond
  arom   -> Bond

  clab(no:int, bond:Bond) -> CLabel
  concLab( CLabel* ) -> CLabelList
  symb(atom:Atom, labels:CLabelList) -> Symbol

  rct(bond:Bond,symbol:Symbol,rctList:ReactantList) -> Reactant
  conc( Reactant* ) -> ReactantList
```

Fig. 11. The signature for TOM terms

We represent a decorated molecular tree as a term of sort `Reactant` as follows:

– a leaf v is a term of sort `Reactant`,

```
rct(b, symb(a, concLab(labs*)), conc())
```

where `a` encodes the label of the leaf (an atom symbol), `b` encodes the label of the edge connecting v with his father, and `labs*` is a possibly empty list of pairs of integers and bond types representing the associated set of broken cycle labels;

– an internal vertex is a term of sort `Reactant`,

```
rct(b, symb(a, concLab(labs*)), conc(rcts*))
```

where `rcts*` encodes the list of its term-like represented children;
– the root has a dummy bond label, `none`, for uniformity reasons.

Operation symbols like `conc` above represent variadic associative operators that construct a list from its arguments (that can be empty).

We consider that a radical point is an atom of valence 1 labelled by `e` (for electron). For efficiency reasons, we consider all free radicals (such as $\bullet x$ in Figure 5) to have tree representations where the electron is the root.

The signatures for GasEl terms and TOM terms are slightly different, but the principles for building the terms are the same. The differences rise from restricting TOM signatures to many-sorted ones, while in ELAN one can use order-sorted signatures. Moreover, the operation symbols in TOM can only be given in prefix notation. These conditions for TOM terms are imposed by the implementation, but in this way, TOM terms are completely explicit.

5.2 Mapping construction

In order to define necessary abstract data-types, we use the signature definition mechanism (`%typeterm`, `%typelist`, `%op`, etc.) provided by TOM.

For example, given a Java class `Reactant`, we can define an algebraic mapping for it:

```
%typeterm Reactant {
  implement { Reactant }
  equals(t1, t2) { t1.equals(t2) }
}
```

where the class `Reactant` has the following structure:

```
class Reactant {
 private Bond bond;
 private Symbol symbol;
 private ArrayList rctlist;

 public Reactant(Bond bond, Symbol symbol, ArrayList rctlist) {...}
 ....
}
```

We can define the following constructor for the `Reactant` type:

```
%op Reactant rct(bond:Bond, symbol:Symbol, radlist:ReactantList) {
  is_fsym(t)                 { t instanceof Reactant }
  get_slot(bond,t)           { t.getBond() }
  get_slot(symbol,t)         { t.getSymbol() }
  get_slot(rctlist,t)        { t.getRctlist() }
  make(bond,symbol,radlist) { new Reactant(bond, symbol, radlist) }
}
```

In fact, this algebraic operation is a mapping from algebraic terms to Java objects that preserves the structural properties of `Reactant` sorted terms for `Reactant` Java instances, i.e. is a *formal anchor*. Let us remind that the formal anchor is determined by the semantics of three mappings: eq, is_fsym, $subterm$. The construct `%typeterm` contains the definition of eq, `equals(t1,t2)`. The other two mapping definitions are given by means of the `%op` construct for the operation symbol `rct`: the mapping $is\_fsym(t, rct)$ is implemented by the construct `is_fsym(t)`, while the mapping $subterm(t, i)$ is implemented by three constructs `get_slot` for retrieving each of the three arguments of `rct`.

Instead of explicitly building this mapping, we can use the two external tools developed together with TOM, Vas and ApiGen, to generate Java files implementing the signature. Doing so, we take advantage of the ATerm library and the VisitableVisitor design pattern which are automatically implemented by the generated classes. The memory sharing is very important for the implementation of reactants because the terms encoding them have in general many common subterms, while the Visitor pattern is necessary for doing term traversals.

The construct `%vas` allows to define a Vas grammar in a TOM file:

```
%vas {
 module data
  imports ...
  public
  sorts Atom Bond CLabel CLabelList Symbol Reactant ReactantList ...
  abstract syntax
  ...
}
```

Considering the signature described by Figure 11, after running Vas , some standard directories are generated which contain all classes that make up the API for the signature. At the root level, the directory contains several standard classes and the mapping for TOM (`data.tom`). The subdirectory `types` contains abstract base classes for each sort defined in the signature, and one subdirectory per sort that contains concrete classes for each operator of this co-arity.

The TOM implementation uses a specialized version of the Visitor design pattern, the VisitableVisitor pattern, based on the *visitor combinators* concept introduced in [28] which allows composition and full tree traversal control. The basic visitor combinators are inspired by the *strategy* primitives of Stratego which are presented in Figure 1 (except the *Omega* strategy). The Java classes generated for the algebraic operations defined within a Vas construct implement the `Visitable` interface. The built-in or user defined traversal strategies are visitable as algebraic terms; on the other side they define `visit_Sort` and `visit_ValueSort_OperationSymbol` methods necessary for visiting algebraic operations.

5.3 Reaction rules

The reaction rules have the form:

$$r : t_1[\,+\,t_2] \rightarrow t_1' + t_2' \text{ if } C$$

(where the elements between square brackets are optional), and we implement them using a match construct according to the following schema:

```
%match(Reactant subject1 [ , Reactant subject2]) {
    t1 [ , t2] → {
            if(C) return pair(σ(t1'), σ(t2'));
    }
}
```

where the argument of match is the term we want to rewrite (the reactant), and σ is the substitution resulted from the matching process. Let us notice that only the implementations of termination rules have two reactants in their left-hand sides.

For all types of reaction rules we define a base class `ChemicalRule` which encloses the common features of all reaction rules. For each reaction application we determine the reaction products and its *degeneration* (how many times the reaction can be applied in different parts of reactants with equal results).

In GasEl one of the implementation difficulties was to have exhaustive application of a reaction rule on one or two reactants. Since the reaction rules are encoded in ELAN as named strategies which can be applied only at the top of a term, exhaustive application in GasEl is achieved by generating all tree-like visions of an acyclic molecular graph (a vision is obtain by choosing a root among the non-hydrogen labelled vertices).

In TOM the solution for this problem is provided in an elegant way by using the strategy *Omega* (Figure 1). Given a term t and a rewrite rule $r : t_1 \rightarrow t_2$, the *Omega* strategy provides the following features:

- we can apply a topdown (or other traversal) strategy for solving the matching problem $t_1 \ll t$; successful matches give rise to a family of substitutions $\{\sigma_i\}_i$;
- for each match solution i, the position ω_i in t where the pattern matched can be retrieved as a Java object by means of the static method `getPosition()` of the class `MuTraveler`;
- for a position ω_i, the subterm $t_{|\omega_i}$ is returned by the method `getSubterm()`;
- for a position ω_i, the term resulting from t after applying the rewriting rule r, i.e. $\sigma_i(t)$ is computed using the method `getReplace`$(\sigma_i(t_2))$.

This is, up to our knowledge, an original feature that provides full control for applying a rewriting rule and allows a wide range of applications. In particular this is quite convenient for applying a reaction rule.

From the implementation point of view, there are two classes of reaction rule: the first class consists of the reactions (ui), (bi), (me), and (ipso) corresponding

to an implementation by topdown traversal of a term in search for a reaction pattern, while the second class consists of the rest of the reactions for which the pattern (with the radical point) is always searched at the root. We illustrate these two types of implementation with the following two examples.

Example 2. [Bimolecular initiation reaction] The generic reaction is:

$$\mathbf{O=O} \quad + \quad \mathbf{H}-x \quad \longrightarrow \quad \bullet\mathbf{OOH} \quad + \quad \bullet x$$

and an application is illustrated in Figure 4. The result of applying the (bi) reaction rule on a term `subject` is implemented by means of the following lines:

```
if( !containsElectron(subject) && (nC(subject) > 1)) {
  VisitableVisitor birule = new BiRule();
  `TopDown(Try((birule))).visit(subject);
  this.setResultList(birule.getResultList());
}
```

First we test if the reactant does not contain a radical point (encoded as an electron), and if it contains at least two carbon atoms. If the test is successful, then we apply in a topdown manner a rule, instance of the class `BiRule`.

For every subterm of sort `Reactant`, during the top-down traversal of the subject of the reaction, the following method of the object `birule` is applied:

```
public Reactant visit_Reactant(Reactant arg) throws VisitFailure {
  Reactant r1, r2;
  int n;
  %match(Reactant arg) {
    rct(b, symb(C(), concLab(labs*)), conc(rcts*)) -> {
      n = nH(arg);
      if( n >= 1) {
        Position pos = MuTraveler.getPosition(this);
        r1 = insertElectron(pos.getSubterm().visit(globalSubject));
        r2 = hangE(pos.getReplace(r1).visit(globalSubject));
        addMPack(`mpack(n, pack(ctRcts.eoo, ctRcts.seoo),
                              pack(r2, usmiles(r2))));
      }
    }
  }
  return `Fail().visit(arg);
}
```

The variable `globalSubject` is set to the value of the term participating to the reaction. We search within the term for a non-aromatic carbon atom which has at least one hydrogen bound by examining all subterms of sort `Reactant`. `nH` computes the number of hydrogen atoms connected to the **C** atom.

We attach an electron to the found carbon atom, we insert the new term in the context, and then we twist the term by means of `hangE` such that node labelled by `e` becomes the root in the corresponding molecular tree in order to preserve the chosen representation of free radicals.

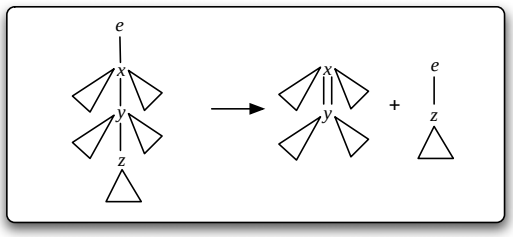

Fig. 12. Beta-scission on terms

A term of sort `Pack` represents a pair composed of a `Reactant` term and its "à-la-smiles" canonical form computed with an algorithm presented in [30]; `eoo` is a constant term corresponding to $\bullet$**OO**, while `seoo` is the canonical form of `eoo`; `n` is the degeneration of the reaction. The method `addMPack` adds an element consisting in a pair of `Pack`-sorted terms with the multiplicity `n` to a private list of this type; this list represents the result of the exhaustive application of a particular reaction rule.

Example 3. [Beta-scission reaction with no cycle breaking] The generic reaction is:

$$\bullet x - y - z \quad \longrightarrow \quad x = y \quad + \quad \bullet z$$

This reaction rule described by subgraphs is easily translated in a rule over trees (as we can see schematically in Figure 12) which is matched at the top of a term (because the electron is always placed in the root).

5.4 Reactor Strategy

We present in this section the implementation of reactor dynamics formally described by the algorithms in Figures 8 and 9. We implement the function UNIT given in Figure 7 by means of the visitor class `UnitRule` with a private member consisting of an array of chemical rules:

```
class UnitRule extends data.dataVisitableFwd {
  private Object rules[];
  public UnitRule(Object rules[]) {
    super(‘Fail());
    this.rules = rules;
  }
  public PairPackList visit_PairPackList(PairPackList arg) { ... }
  public PackList visit_PackList(PackList arg)  { ... }
}
```

`UnitRule` can be used as a rule with a particular behavior on terms of sorts `PairPackList` and `PackList`. Each of the *visit_Sort* methods contains applications of the rules passed as arguments on lists of reactants.

The initiation stage described with the algorithm $UNIT(R_i)$ in Figure 8 is implemented as:

```
ChemicalRule initRules[] =
             {new UICCRule(), new UICHRule(), new BiRule()};
VisitableVisitor init_unit  = new UnitRule(initRules);
plist = `Try(init_unit).visit(plist);
```

where `plist` from the right-hand side is the input list of chemical reactants (the initial set of reactants), while `plist` from the left-hand side contains the products obtained from the initiation stage together with the input reactants.

For the propagation stage, chemical hypotheses impose to apply the reactions (me) and (ipso) only on the products resulted from the initiation stage. Therefore we describe the propagation stage by means of the strategy $UNIT(R_p); repeat(UNIT(R_p - \{(me), (ipso)\}))$, and we implement it as follows:

```
ChemicalRule propagRules1[] = {new MeRule(), new IpsoRule(),
  new BSCCRule(), new BSCHRule(), new OxRule(), new CombeOeRule()};
VisitableVisitor propag_unit1 = new UnitRule(propagRules1);
tmplist = `Try(propag_unit1).visit(plist);

tmplist = diff(tmplist, plist);
plist = appendLists(plist, tmplist);
pairlist = `pair(plist, tmplist);

ChemicalRule propagRules2[] = {new BSCCRule(), new BSCHRule(),
                               new OxRule(), new CombeOeRule() };
VisitableVisitor propag_unit2 = new UnitRule(propagRules2);
pairlist = `RepeatId(Try(propag_unit2)).visit(pairlist);
plist = getFirstList(pairlist);
```

First we put the reaction products from all propagation rules in `tmplist`, then we select only the free radicals not already in the input list, and put them together with the initial reactants. We make a pair of list with first element consisting in all reactants, and the second element consisting in the list of new free radicals, and we provide it as input for the strategy that applies the chemical rules from the array `propagRules2`. This application of this strategy ends when the list of new free radicals is empty. The result of the propagation stage consists in the list of all resulted chemical elements concatenated with the list of input reactants.

The termination stage described by $UNIT(R_t)$ is implemented in TOM as follows:

```
ChemicalRule terminRules[] = {new CoRule(), new DiRule()};
VisitableVisitor termin_unit  = new UnitRule(terminRules);
plist = `Try(termin_unit).visit(plist);
```

6 Conclusion

The first output of this work is a new prototype of a chemical reactor. First results revealed good properties with respect to chemical validations of the model. A complete comparison between the GasEl prototype and the current implementation in TOM is currently under development. Due to notation and implementation differences, this comparison is not trivial and out of the scope of this paper.

It may be worth noticing that the rule-based approach on graph structures has also been studied in the modelling of signal transduction networks [12] and metabolic pathways [22] in the domains of biological systems and protein interactions. Our model of chemical reactor seems to be easily adaptable to these domains.

The second concern in this work was to explore the formal island concept and methodology on a significant example. The objective of the formal island approach to extend the expressivity of the host language with higher-level constructs at design time is well-illustrated in this example. From this point of view, the TOM implementation appeared to be quite convenient to implement chemical rules with conditions and actions expressed in the Java host language. On the other hand, control was expressed with a high-level language of strategies which makes now possible to reason about formal properties, especially the termination property of each phase [3]. This illustrates the idea to perform formal proof on the formal island constructions.

A further idea would be to implement a new version of the TOM compiler able to perform graph rewriting. Representing cyclic structures in TOM is not too difficult but matching and rewriting have to be adapted to this context. Indeed this capability would open new application areas.

A long-term objective of the formal island approach is to certify the implementation of the formal island compilation into the host language. A first step in this direction has been presented in [18] to generate proof obligations for the compilation of matching. A similar concern is underway for rewriting and strategies.

Further improvements of the formal island approach is to anchor other language extensions, especially modules and parameters, while improving the capacity of the compiler to generate verification requirements related to properties to be checked.

Acknowledgements: We sincerely thank Olivier Bournez, Pierre-Etienne Moreau and Antoine Reilles for helpful remarks and interactions on this work, and the Protheo group for scientific and financial support. This work has been partially funded by an INRIA international internship program.

References

1. Elan web site. `http://elan.loria.fr`.
2. Tom web site. `http://tom.loria.fr`.

3. O. Andrei. Term graph and chemical rewriting. Internship report, LORIA, Protheo Team, Nancy, France, September 2005.
4. E. S. Blurock. Reaction: System for Modeling Chemical Reactions. *Journal of Chemical Information and Computer Science*, 35:607–616, 1995.
5. P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An Overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings of the Second International Workshop on Rewriting Logic and Applications*, volume 15, `http://www.elsevier.nl/locate/entcs/volume15.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science. Rapport LORIA 98-R-316.
6. O. Bournez, G.-M. Côme, V. Conraud, H. Kirchner, and L. Ibănescu. A Rule-Based Approach for Automated Generation of Kinetic Chemical Mechanisms. In R. Nieuwenhuis, editor, *Proceedings of the 14th International Conference on Rewriting Techniques and Applications, RTA 2003, Valencia, Spain, June 9-11, 2003*, volume 2706 of *Lecture Notes in Computer Science*, pages 30–45. Springer, 2003.
7. O. Bournez, L. Ibanescu, and H. Kirchner. From Chemical Rules to Term Rewriting. In *6th International Workshop on Rule-Based Programming*, To appear in ENTCS series, Nara, Japan, April 2005.
8. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, (285):187–243, August 2002.
9. G.-M. Côme. *Gas-Phase Thermal Reactions. Chemical Engineering Kinetics*. Kluwer Academic Publishers, 2001.
10. R. Diaconescu and K. Futatsugi. An overview of CafeOBJ. In C. Kirchner and H. Kirchner, editors, *Electronic Notes in Theoretical Computer Science*, volume 15. Elsevier, 2000.
11. J. Dugundji and I. Ugi. An Algebraic Model of Constitutional Chemistry as a Basis for Chemical Computer Programs. *Topics in Current Chemistry*, 39:19–64, 1973.
12. J. Faeder, M. Blinov, and W. Hlavacek. Graphical rule-based representation of signal-transduction networks. In *ACM Symposium on Applied Computing*, pages 133–140, 2005.
13. J. M. Grenda, I. Androulakis, A. M. Dean, and W. H. Green. Application of Computational Kinetic Mechanism Generation to Model the Autocatalytic Pyrolysis of Methane. *Industrial & Engineering Chemistry Research*, 42:1000–1010, 2003.
14. J. Guyon, P.-E. Moreau, and A. Reilles. An integrated development environment for pattern matching programming. In B. Barry and O. de Moor, editors, *Proceedings of the 2nd eclipse Technology eXchange workshop, eTX'2004 (Barcelona, Spain)*, Barcelona (Spain), 2004. Electronic Notes in Theoretical Computer Science.
15. L. Ibanescu. *Programmation par règles et stratégies pour la génération automatique de mécanismes de combustion d'hydrocarbures polycycliques*. Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, Nancy, France, June 2004.
16. C. Kirchner and H. Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.
17. C. Kirchner and H. Kirchner. Rule-based programming and proving: the ELAN experience outcomes. In *Proceedings of the Ninth Asian Computing Science Conference ASIAN'04*, volume 3371, pages 363–379, Chiang Mai, Thailand, December 2004. Lecture Notes in Computer Science.

18. C. Kirchner, P.-E. Moreau, and A. Reilles. Formal validation of pattern matching code. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 187–197, New York, NY, USA, 2005. ACM Press.
19. H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: A compiler for non-deterministic rewrite programs in associative-commutative theories. *Journal of Functional Programming*, 11(2):207–251, 2001.
20. P.-E. Moreau, C. Ringeissen, and M. Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.
21. E. Ranzi, T. Faravelli, P. Gaffuri, and A. Sogaro. Low-Temperature Combustion: Automatic Generation of Primary Oxidation Reactions and Lumping Procedures. *Combustion and Flame*, 102:179–192, 1995.
22. F. Roselló and G. Valiente. Analysis of metabolic pathways by graph transformation. In H. E. et al., editor, *2nd International Conference on Graph Transformation - ICGT'04, Roma, Italy*, volume 3256 of *Lecture Notes in Computer Science*, pages 70 – 82. Springer, 2004.
23. A. S. Tomlin, T. Turányi, and M. J. Pilling. *Mathematical Tools for the Construction, Investigation and Reduction of Combustion Mechanisms*, volume 35 of *Comprehensive Chemical Kinetics*, chapter 4, pages 293–437. Elsevier, Amsterdam, 1997.
24. M. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Computational Complexity*, pages 365–370, 2001.
25. M. G. J. van den Brand, H. A. de Jong, and P. Olivier. Efficient annotated terms. Technical report, University of Amsterdam, 2000. SEN-R0003, ISSN 1386-369X.
26. E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.
27. E. Visser, Z.-e.-A. Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. *ACM SIGPLAN Notices*, 34(1):13–26, January 1999. Proceedings of the International Conference on Functional Programming (ICFP'98).
28. J. Visser. Visitor combination and traversal control. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA'01, Tampa Bay, Florida, USA*, volume 36(11) of *ACM SIGPLAN Notices*, pages 270–282, 2001.
29. V. Warth, F. Battin-Leclerc, R. Fournet, P.-A. Glaude, G.-M. Côme, and G. Scacchi. Computer Based Generation of Reaction Mechanisms for Gas-Phase Oxidation. *Computers and Chemistry*, 24:541–560, 2000.
30. D. Weininger, A. Weininger, and J. L. Weininger. SMILES. 2. Algorithm for Generation of Unique SMILES Notation. *Journal of Chemical Information and Computer Science*, 29:97–101, 1989.
31. M. J. D. Witt, D. D. Dooling, and L. J. Broadbelt. Computer Generation of Reaction Mechanisms Using Quantitative Rate Information: Application to Long-Chain Hydrocarbon Pyrolysis. *Industrial & Engineering Chemistry Research*, 39(7):2228–2237, 2000.