A Symbolic Transformation Language and its Application to a Multiscale Method

Walid Belkhir, Alain Giorgetti, Michel Lenczner

A Symbolic Transformation Language and its Application to a Multiscale Method

Walid Belkhir [a], Alain Giorgetti [a,b], Michel Lenczner [c]

[a] *INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-lès-Nancy, France*

[b] *FEMTO-ST institute, Département d'Informatique des Systèmes Complexes, University of Franche-Comté 16 route de Gray, 25030 Besançon Cedex, France*

[c] *FEMTO-ST institute, Département Temps-Fréquence, University of Technology of Belfort-Montbéliard 26 chemin de l'Epitaphe, 25030 Besançon Cedex, France*

Abstract

The context of this work is the design of a software, called *MEMSALab*, dedicated to the automatic derivation of multiscale models of arrays of micro- and nanosystems. In this domain a model is a partial differential equation. Multiscale methods approximate it by another partial differential equation which can be numerically simulated in a reasonable time. The challenge consists in taking into account a wide range of geometries combining thin and periodic structures with the possibility of multiple nested scales.

In this paper we present a transformation language that will make the development of MEMSALab more feasible. It is proposed as a Maple™ package for rule-based programming, rewriting strategies and their combination with standard Maple™ code. We illustrate the practical interest of this language by using it to encode two examples of multiscale derivations, namely the two-scale limit of the derivative operator and the two-scale model of the stationary heat equation.

Key words: Symbolic transformation, term rewriting, strategies, multiscale modeling.

1. Introduction

The context of this work is the design of microsystem array architectures, including microcantilevers, micromirrors, droplet ejectors, micromembranes, microresistors, etc.,

⋆ This work has been partially funded by the Labex ACTION (ANR-11-LABX-01-01), the INTERREG IV project OSCAR and the PPF MIDi.

Email addresses: walid.belkhir@inria.fr (Walid Belkhir), alain.giorgetti@femto-st.fr (Alain Giorgetti), michel.lenczner@utbm.fr (Michel Lenczner).

to cite only a few. A model for such arrays is a Partial Differential Equation (PDE). The numerical simulation of whole arrays based on classical methods like the Finite Element Method (FEM) is prohibitive for today's computers (at least in a time compatible with the time scale of a designer). The calculation of a reasonably complex cell of a three-dimensional microsystem requires at least 1000 degrees of freedom which lead to at least 10 000 000 degrees of freedom for a 100×100 array. Fortunately there is a solution consisting in approximating the model by a multiscale method. The resulting approximated model is again a PDE. It can be rigorously derived from the exact one through a sequence of mathematical transformations, but these transformations differ for each case.

We are currently developing a software, called MEMSALab, for "MEMS Arrays Laboratory", dedicated to multiscale and multiphysics modeling of arrays of micro- and nanosystems. Unlike traditional software that is based on models built once and for all, MEMSALab is a software that constructs models. The challenge consists in taking into account a wide range of geometries combining thin and periodic structures with the possibility of multiple nested scales. One should also consider PDEs representing multiphysics systems with high contrast in equation coefficients.

Simulation software available in the market offers specialized tools for large arrays of micro- and nanosystems, but the construction of new models raises many problems. Firstly the time required for a new design varies from some weeks for a specialist to several months for a beginner. Secondly the mathematical machinery is too sophisticated to be manually applied to complex systems. Finally the resulting models require specific numerical simulation methods, that have to be implemented case by case.

The software MEMSALab we design aims at addressing these problems. It is based on multiscale models, especially on those derived by asymptotic methods. Such asymptotic models are derived from a system of PDEs when taking into account that at least one parameter is very small, such as thickness for a thin structure or the small ratio of a cell size to the global size for a periodic structure. The resulting models are other systems of PDEs, obtained by taking the mathematical limits of the nominal models, in a well-suited sense, when the small parameters tend toward zero. This approach provides a reasonably good approximation. It also offers the advantages and factors of reliability to be rigorous and systematic. The resulting PDEs can be implemented in a simulation software such as the finite element based simulator COMSOL (Multiphysics Finite Element Analysis Software, official site `http://www.comsol.com`), and simulations turn to be fast as needed.

The literature in this field is vast and a large number of techniques have been developed for a large variety of geometric features and physical phenomena. However, none of them have been implemented in a systematical approach to render it available to engineers as a design tool. In fact, each published paper focus on a special case regarding geometry or physics, and very few works are considering a general picture. By contrast our software will treat the problem of systematic implementation of asymptotic methods by implementing the construction of models rather than the models themselves. This approach will cover many situations from a small number of bricks. It combines mathematical and computer science tools. The mathematical tool is the two-scale transform originally introduced in (Lenczner, 1997; Casado-Díaz, 2000; Cioranescu et al., 2002) to model periodic and thin structures, and also referred as the unfolding method. We have extended its domain of application to cover in the same time homogenization of periodic media, see for instance Bensoussan et al. (1978), and methods of asymptotic analysis

for thin domains, see Ciarlet (1988). The computer science tools include term rewriting, λ-calculus, and type systems (Cirstea and Kirchner, 2001; Marin and Piroi, 2004; Cirstea et al., 2001; Geuvers, 2009). The software is written in the symbolic computation language Maple™.

Compared to other techniques, our multiscale method requires more modular calculations, avoids any non-constructive proof and intensively relies on equational reasoning. The classical way to automate equational reasoning is to consider mathematical equalities as rewrite rules. The rewrite rule $t \to u$ orients the equality $t = u$ from left to right and states that every occurrence of an instance of t can be replaced with the corresponding instance of u. Consequently symbolic computation with equalities is reduced to a series of term rewritings. Algebraic computation and term rewriting are two research domains with strong similarities. Both are separately well-studied but there are only few works about the combination of algebraic computation and term rewriting (Fèvre and Wang, 1998; Bündgen, 1995). Term rewriting provides a theoretical and computational framework which is very useful to express, study and analyze a wide range of complex systems. It is characterized by a repeated transformation of data objects such as words, terms or graphs. Transformations are described by a combination of rules which specify how to transform an object into another one in the presence of a specific pattern. Rules can have further conditions and can be combined by specifying strategies. The latter control the order and the way the rules are applied. Term rewriting is used in semantics in order to describe the meaning of programming languages as well as in program transformations. It is used to perform symbolic computations like in Mathematica, and also to perform automated reasoning. It is central in systems where the notion of *rule* is explicit such as expert systems, algebraic specifications, etc.

The computer algebra system Maple™ is widely used in the symbolic computation community. It is also used by members of our project for a prototypal implementation of MEMSALab. Maple™ is a suitable language for combining function-based and rule-based symbolic transformations. Unfortunately, it is only equipped with a limited rewrite kernel, namely the function `applyrule(rule,expr)`. The main drawback of `applyrule` is that it iterates the application of the rule everywhere in the given expression until no matching sub-expression remains. Therefore there is a lack of flexibility: the user cannot express how and where rules must be applied. The other problem is that the Maple™ matching function `patmatch(expr,pattern,sub)` has two main drawbacks. Firstly, when the user defines new operators, the matching must be done modulo the properties of these operators, e.g. associativity and commutativity. Actually, `patmatch` does not provide this feature. Secondly the matching function computes just one solution (i.e. a substitution) of the matching problem of `pattern` with `expr`, whereas both supporting associative-commutative operators and implementing conditional rewrite rules require a matching that returns all the possible solutions.

Contributions

In this work we present a transformation language named symbtrans (for "symbolic transformation"). It extends Maple™ with conditional rewriting, strategies, and pattern-matching modulo associativity and commutativity. All the ingredients of term rewriting with strategies are made explicit. In particular terms, patterns, rules, strategies, pattern matching and application of rules and strategies to terms are represented by Maple™ expressions. Rewrite rules and rewriting strategies are deterministic functions that raise

an exception when they are not applicable. Such functions can be combined with Maple™ functions. We illustrate the practical interest of this transformation language by using it to write two formal derivations for the MEMSALab software. The first one states the weak two-scale limit of the derivative operator. The second one computes the two-scale model of the stationary heat equation.

Related work

Our transformation language is an adaptation for Maple™ of popular strategy languages such as ρ-log (Marin and Piroi (2004)) or Tom (Balland et al. (2007)). A conceptual difference with Tom is that the latter extends a host language with an additive syntax, whereas our transformation language smoothly integrates with standard Maple™ functions. The first work that have considered term rewriting from a functional point of view is Elan, see (Borovansky et al., 2001), within a non-deterministic framework. Our transformation language is comparable with the deterministic fragment of the *rewriting calculus* (Cirstea and Kirchner, 2001).

Paper outline

Section 2 introduces the two-scale transform and necessary term rewriting concepts and notations. Section 3 defines the transformation language symbtrans in a detailed way and illustrates transformations by several examples. Section 4 presents more advanced features of symbtrans: the combination of term rewriting with procedural programming, a delayed procedure evaluation mechanism, pattern-matching modulo associativity and commutativity, and conditional rewriting. Section 5 shows the transformation language at work on two realistic examples of multiscale derivations. Section 6 presents the theoretical bases of symbtrans and compares the present work with related ones. Section 7 draws conclusions.

A mathematical proof of the weak two-scale limit property of the derivative operator is reproduced in Appendix A. Its formal proof with symbtrans is reproduced in Appendix C. A mathematical derivation of the two-scale model of the stationary heat equation is reproduced in Appendix B. Its formal counterpart with symbtrans is given in Appendix D. Rules and transformations corresponding to mathematical properties for these formal proofs are reproduced in Appendix E.

2. Preliminaries and notations

2.1. Term, substitution, matching and rewriting

Term rewriting systems (in the classical sense) are defined by specifying a set of terms and a set of rewrite rules. Rewrite rules are applied to *reduce* terms. In general the process of reduction continues until no more rules can be applied, or forever in the case of non-terminating systems. A term which cannot be reduced to another term is called a *normal form*. If there is always a unique normal form then the system is said to be *confluent*.

Let $\mathcal{F}$ be a countable set of function symbols, each symbol having a fixed arity. Let $\mathcal{X}$ be a countable set of variables. The set of terms, denoted by $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is inductively defined as the smallest set containing the elements of $\mathcal{X}$ and $f(t_1, \ldots, t_n)$, for any symbol

f with arity n in $\mathcal{F}$ and any terms $t_1, \ldots, t_n$ in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. When $n = 0$ the symbol f is called a *constant* and the corresponding term is denoted f instead of $f()$.

The set of variables occuring in a term t is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t) = \emptyset$ then t is said to be *ground*. A *substitution* is a function σ from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $\sigma(x) \neq x$ only for finitely many variables x in $\mathcal{X}$. If $x_1, \ldots, x_n$ are these variables, then σ is denoted by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $t_i = \sigma(x_i)$ $(1 \leq i \leq n)$. If t is a term then $\sigma(t)$ is the term that results from the application of σ to t.

A *rewrite rule* is a pair (l, r) of terms l and r in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ s.t. $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. This pair is usually written $l \to r$.

Definition 1. For two terms t and t' in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the problem of finding substitutions σ such that $\sigma(t') = t$ is called a *matching problem* and is denoted $t' \ll t$. A substitution σ such that $\sigma(t') = t$ is called a *solution* of this matching problem. The *application* of a rewrite rule $l \to r$ to a term t, denoted by $[l \to r](t)$, is defined by $\sigma(r)$ where σ is any solution of the matching problem $l \ll t$. It is undefined when this matching problem has no solution.

Example 2. Let f, g , $+$ and $*$ (resp. a and b) be function symbols of arity 2 (resp. 0) in $\mathcal{F}$. Let x and y be variables in $\mathcal{X}$.

- The application $[x \to b](f(a, b))$ of the rule $x \to b$ to the term $f(a, b)$ is the term b. The substitution $\sigma = \{x \mapsto f(a, b)\}$ is an obvious solution of the matching problem $x \ll f(a, b)$.
- The application $[f(x, a) \to g(a, x)](f(b, a))$ of the rule $f(x, a) \to g(a, x)$ to the term $f(b, a)$ is the term $g(a, b)$, with the substitution $\sigma = \{x \mapsto b\}$.
- The application $[a \to b](a)$ of the rule $a \to b$ to the term a is the term b, with the empty substitution $\sigma = \{\}$.

It is worth mentioning that there is a difference between the notions of variables and constants in the mathematical sense and in the sense of term rewriting when mathematical expressions are viewed as terms. Let us give a concrete example. Consider the mathematical expression $\int_\Omega f(x)dx$ and its representation by the term `Integral(Omega, f(x), x)`. In this term, `Omega` and the mathematical variable `x` are viewed as function symbols of arity 0 (i.e. constants), `f` is viewed as a function symbol of arity 1, and `Integral` as a function symbol of arity 3. Further clarifications on this topic can be found in Section 3.

2.2. Running example

We illustrate the transformation language with the homogenization problem of the stationary heat equation. We consider a stationary distribution of temperature in a region $\Omega \subset \mathbb{R}^n$ (where $n \in \{1, 2, 3\}$) with an internal heat source and with imposed vanishing temperature along the boundary. The diffusion coefficient $a^\varepsilon : \mathbb{R}^n \to \mathbb{R}$ is assumed to be periodic on Ω in the n directions, with a small period ε. In other words, there is a function $a : \mathbb{R}^n \to \mathbb{R}$ which is $(0, 1)^n$-periodic and such that $a^\varepsilon(x) = a(x/\varepsilon)$ for $x \in \Omega$. With a view to derive the so-called *homogenized model*, the parameter ε is considered as small, and we are interested in finding an approximation of the stationary heat equation when ε decreases to zero. In this mathematical asymptotic process, the distributed internal heat source $f^\varepsilon : \mathbb{R}^n \to \mathbb{R}$ can be considered as depending on ε, and the temperature

distribution $u^\varepsilon : \mathbb{R}^n \to \mathbb{R}$, vanishing on the boundary $\partial\Omega$ of Ω, is the unique solution to the stationary heat equation

$$\sum_{i=1}^{n} \int_\Omega a^\varepsilon \partial_{x_i} u^\varepsilon \, \partial_{x_i} v \; dx = \int_\Omega f^\varepsilon v \; dx \tag{1}$$

written in its variational form, as explained in (Dautray and Lions, 1990). Here, $v : \Omega \to \mathbb{R}$ is any *test function*, i.e. a sufficiently regular function vanishing on $\partial\Omega$.

To conduct the asymptotic process $\varepsilon \to 0$ while keeping as much information as possible in the solution u^ε at the small scale, the region Ω is unfolded into the Cartesian product $\widetilde{\Omega} \times Y$ with $Y = (0,1)^n$ through the so-called *two-scale transform* or *unfolding*, where $\widetilde{\Omega} \subset \mathbb{R}^n$ is called the *macroscopic domain* associated to Ω, and Y is called the *microscopic domain* associated to Ω. This sort of change of variables is applied to the sequence $u^\varepsilon : \Omega \to \mathbb{R}$ which yields another sequence of functions $Tu^\varepsilon : \widetilde{\Omega} \times Y \to \mathbb{R}$. We can show that the latter converges to a limit $u^0(x,y)$, so we say that the sequence u^ε is *two-scale convergent* to u^0. Similarly, we show that a^ε, f^ε and $\partial_{x_i} u^\varepsilon$ are two-scale convergent towards some limits a^0, f^0 and $\partial_{x_i} u^0 + \partial_{y_i} u^1$, and in the same time that u^0 is independent of the so-called *microscopic variable* y, it is vanishing on $\partial\widetilde{\Omega}$ and u^1 is Y-periodic, a concept that is explained later. Then, applying the two-scale transform in the variational formulation (1) of the heat equation, passing to the limit $\varepsilon \to 0$, we arrive to the *two-scale model* satisfied by the pair (u^0, u^1),

$$\sum_{i=1}^{n} \int_{\widetilde{\Omega}\times Y} a^0 \left(\partial_{x_i} u^0 + \partial_{y_i} u^1\right) \left(\partial_{x_i} v^0 + \partial_{y_i} v^1\right) dxdy = \int_{\widetilde{\Omega}\times Y} f^0 \, v^0 \; dxdy, \tag{2}$$

where the pair $(v^0(x), v^1(x,y))$ is sufficiently regular, v^0 is vanishing on $\partial\widetilde{\Omega}$ and v^1 is Y-periodic. The last step consists in showing that u^1 is a function of u^0 which allows its elimination and yields the *homogenized model* [1]

$$\sum_{j,k=1}^{n} \int_{\widetilde{\Omega}} a^H_{jk} \partial_{x_j} u^0 \, \partial_{x_k} v^0 dx = \int_{\widetilde{\Omega}} f^H v^0 dx \tag{3}$$

for all test functions v^0. Here a^H is the $n \times n$ matrix of *effective diffusion coefficients*, and f^H is the *effective heat source*. We observe that in this example the macroscopic domain $\widetilde{\Omega}$ is identical to Ω. However, we prefer to distinguish them for sake of generalization, e.g. to thin domains.

The statement of the approximate model (3) derived with the two-scale transform was announced in (Lenczner, 1997). Then several proofs have been published, in (Lenczner and Senouci-Bereksi, 1999), (Casado-Díaz, 2000), (Cioranescu et al., 2002), (Lenczner and Smith, 2007), and (Cioranescu et al., 2008). Here we follow the proof in (Lenczner and Smith, 2007) where an effort has been made to formulate proofs in an algebraic way and to avoid any abstract reasoning in the sequences of formal transformations. The steps in this formal method are rigorously specified at a high level of generality, that make them independent of the domain geometry and applicable to other equations.

[1] We mention that we use the minimal scope convention of the derivative operator, i.e. $\partial_{x_i} fg$ means $(\partial_{x_i} f)g$.

Before entering in more details, we introduce some notations and definitions. For any region $\Theta \subset \mathbb{R}^n$, $L^2(\Theta)$ denotes the set of square integrable functions on Θ, that is the set of functions $v : \Theta \to \mathbb{R}$ with a bounded $L^2(\Theta)$-norm

$$||v||_{L^2(\Theta)} = (\int_\Theta v^2(x)\ dx)^{1/2}.$$

Then a sequence $u^\varepsilon \in L^2(\Theta)$ is said to be convergent (or strongly convergent) in $L^2(\Theta)$ towards a limit u^0 if $||u^\varepsilon - u^0||_{L^2(\Theta)} = O(\varepsilon)$ where $O(\varepsilon)$ is the Landau notation representing any sequence of ε tending to zero when $\varepsilon \to 0$. Another concept of convergence used for asymptotic models is the concept of weak convergence. A sequence $u^\varepsilon \in L^2(\Theta)$ is said to be weakly convergent in $L^2(\Theta)$ towards a limit u^0 if

$$\int_\Theta (u^\varepsilon - u^0)v\ dx = O(\varepsilon) \text{ for any } v \in L^2(\Theta).$$

A by product of this definition is that a sequence $u^\varepsilon \in L^2(\Omega)$ is said to be *two-scale weakly convergent* towards a limit $u^0 \in L^2(\widetilde{\Omega} \times Y)$ if Tu^ε is weakly convergent towards u^0 in $L^2(\widetilde{\Omega} \times Y)$. Finally, a function $v : Y \to \mathbb{R}$ is said to be Y-periodic if $v(y^+) = v(y^-)$ for any pair of opposite points y^+ and y^- of the boundary of Y.

Now we specify the assumptions needed to build the model (3) as the limit of (1). The heat source f^ε is assumed to be uniformly bounded in the $L^2(\Omega)$-norm, that is, there exists a constant C independent of ε such that

$$||f^\varepsilon||_{L^2(\Omega)} \leq C. \tag{4}$$

Then, the proof is divided into five parts. (i) We establish that the fields of temperature distribution u^ε and its derivatives are uniformly bounded in the $L^2(\Omega)$-norm,

$$||u^\varepsilon||_{L^2(\Omega)} \leq C \text{ and } ||\partial_{x_i} u^\varepsilon||_{L^2(\Omega)} \leq C \text{ for } i = 1, ..., n. \tag{5}$$

Given these results, it is assumed that u^ε has an asymptotic expansion on the form

$$Tu^\varepsilon = u^0 + \varepsilon \widetilde{u}^1 + \varepsilon O(\varepsilon), \tag{6}$$

where here $O(\varepsilon)$ denotes a function that tends to zero weakly in $L^2(\widetilde{\Omega} \times Y)$. We observe that this assumption is not necessary to get the desired result, however, we find that on the one hand it is not very strong once the a priori estimates are known and on the other hand it allows the proof to be entirely computational, i.e. without steps of abstract reasoning. (ii) The next step consists in deducing of (5) that the two-scale weak limit u^0 of Tu^ε is independent of y

$$\partial_{y_i} u^0 = 0 \text{ for all } i. \tag{7}$$

(iii) It comes to show that there exists $u^1(x, y)$ such that

$$\partial_{x_i} u^\varepsilon \text{ is weakly two-scale convergent towards } \partial_{x_i} u^0 + \partial_{y_i} u^1 \text{ in } L^2(\widetilde{\Omega} \times Y), \tag{8}$$

together with the relation between u^1 and $(u^0, \widetilde{u}^1)$,

$$\widetilde{u}^1 = u^1 + \sum_{j=1}^{n} y_j \partial_{x_j} u^0 \tag{9}$$

as well as the fact that u^1 is Y-periodic. (iv) Once the two-scale limit of $\partial_{x_i} u^\varepsilon$ has been computed, it is used in the variational formulation (1) in a manner that yields the

two-scale model (2). (v) Finally, the homogenized model (3) is deduced by expressing each microscopic derivatives $\partial_{y_i} u^1$ as a linear function of the macroscopic derivatives $(\partial_{x_j} u^0)_{j=1,\dots,n}$, the linear operator between them being solution of a partial differential equation in the cell Y.

It would take too long to present in detail all of the above proof. We chose to illustrate the transformation rules only on the third and fourth steps because they require the implementation of most of the useful concepts for the complete proof. In addition, we do not show the periodicity of u^1 nor the relation (9) between u^1 and $(u^0, \widetilde{u}^1)$. Thus, the two examples discussed are to prove the following propositions. The mathematical proofs are reported in Appendices A and B respectively when their counterpart formalized by rewriting rules and strategies are in Appendices C and D.

Proposition 3. *For a sequence of functions $u^\varepsilon : \Omega \to \mathbb{R}$ such that u^ε and $\partial_{x_i} u^\varepsilon$ are bounded in the $L^2(\Omega)$-norm, if Tu^ε has a formal expansion of the form*

$$Tu^\varepsilon = u^0 + \varepsilon u^1 + \varepsilon \sum_{j=1}^{n} y_j \partial_{x_j} u^0 + \varepsilon O(\varepsilon), \tag{10}$$

where u^0 is independent of y, the partial function $y \mapsto u^1(., y)$ is Y-periodic and $O(\varepsilon)$ tends to zero in $L^2(\widetilde{\Omega} \times Y)$ weak, then (8) holds.

Proposition 4. *Assuming that the data of the stationary heat equation (1) satisfy*

$$Ta^\varepsilon = a^0 \text{ and } Tf^\varepsilon = f^0 + O(\varepsilon), \tag{11}$$

where $O(\varepsilon)$ tends to zero in $L^2(\widetilde{\Omega} \times Y)$ and the solution u^ε satisfies the assumptions and the conclusion of Proposition 3, then the pair (u^0, u^1) is solution to the two-scale model (2).

A number of mathematical tools are required to carry out the proofs. Some of them are refered in the body of the paper so they are described in the end of this section while the others are used only in the detailed proofs and thus are presented in the beginning of Appendix A.

In order to formalize the convergence concepts of sequences of numbers and of functions, and to handle them within a computational framework, we recall the usual set of computation rules of the Landau notation $O(\varepsilon)$ and we also add redundant rules for sums and integrals. As usual the computation rules on the Landau notation apply from left to right only.

$$\begin{aligned} &O(\varepsilon) + O(\varepsilon) = O(\varepsilon), \;\; -O(\varepsilon) = O(\varepsilon), \;\; \sum O(\varepsilon) = O(\varepsilon), \;\; \int O(\varepsilon)\, dx = O(\varepsilon), \\ &O(\varepsilon) * O(\varepsilon) = O(\varepsilon), \;\; \alpha * O(\varepsilon) = O(\varepsilon) \text{ if } \alpha \text{ is independent of } \varepsilon, \text{ and } \varepsilon = O(\varepsilon). \end{aligned} \tag{12}$$

This small system of "*axioms*" defines what we call the *convergence calculus* for the rewriting rules. We observe that it does not include a mean for distinguishing the various types of convergences, so for now that distinction is left to the user. We shall also repeatedly use the additional property of $O(\varepsilon)$ whereby

$$\int_\Omega g\; O_1(\varepsilon)\; dx = O(\varepsilon) \tag{13}$$

as long as g is a function uniformly bounded in $L^2(\Omega)$ and $\lim_{\varepsilon\to 0} ||O_1(\varepsilon)||_{L^2(\Omega)} = 0$. It results from the Cauchy-Scwharz inequality $\int_\Omega g\ O_1(\varepsilon)\ dx \leq ||g||_{L^2(\Omega)}||O_1(\varepsilon)||_{L^2(\Omega)} \leq C\ O(\varepsilon) = O(\varepsilon)$.

The usual operations on variational formulation require to use the extension of the rule of integration by parts to multidimensional domains. The so-called *Green formula* holds for sufficiently regular functions u and v defined in a domain $\Theta \subset \mathbb{R}^n$,

$$\int_\Theta u\ \partial_{x_i} v\ dx = \int_{\partial\Theta} u\ v\ (n_x)_i\ ds(x) - \int_\Theta v \partial_{x_i} u\ dx \tag{14}$$

for any $i = 1, ..., n$ where n_x represents the outward pointing unit normal of the hypersurface volume element $ds(x)$ on the boundary $\partial\Theta$.

Finally, functions such as u and v are usually considered as elements of a vector space such as $L^2(\Omega)$. As such, it is possible to define linear operators that apply to them. We recall that an operator L defined on a vector space is said to be *linear* if for any vectors v, w and any scalar α,

$$L(\alpha\ v) = \alpha\ L(v) \tag{15}$$

$$\text{and } L(v + w) = L(v) + L(w). \tag{16}$$

3. Transformation language

This section defines a transformation language based on the three notions of *rule*, *strategy* and *transformation*. By a rule we mean a classical rewrite rule. A strategy is a way to control how rules are applied. Strategies can be combined to define strategies with a finer control or a more powerful effect. We propose easy-to-remember names for the most popular – and indeed most useful – strategy constructors and combinators. The user can also extend the language with other strategies.

It is not obvious that any natural transformation of mathematical expressions and models can be concisely expressed as a rewriting-based strategy, in a natural way. Moreover what is exactly a strategy is not completely clear from the literature, and the name of "strategy" for a formal transformation may lead the user of our language to confusion. Therefore we consider an a priori independent notion of *transformation*. Transformations have the following three features: (i) a transformation is reproducible, (ii) a transformation may not progress, and (iii) a transformation may not terminate. The reproducibility feature (i) means that each application of a given transformation to a given expression produces the same effect: either it does not terminate each time, or it produces each time the same expression. In particular, this property excludes non-determinism from the notion of transformation. By contrast a strategy may be non-deterministic (see, e.g. `One[s]` in Balland et al. (2007)). It would be enriching to develop a complete theory of transformations but it exceeds the scope of the present paper on rule-based transformations.

We propose an implementation of this transformation language as a new package for the Maple™ computer algebra system. The package is named symbtrans, for "symbolic transformations".[2] We strongly rely on the functional features of Maple™ by providing rules, strategies and transformations as Maple functions, possibly through higher-order

[2] The package is available as an archive file upon request to the authors. It can be executed on any machine running Maple™.

functions constructing them from another representation. Functions faithfully provide the expected feature (i) of transformation reproducibility. Whether a transformation output differs or not from its input (feature (ii)) is controlled by the Maple™ mechanism of exceptions. Feature (iii) is left under the responsability of users that can interrupt execution with the Maple™ function `timelimit`.

3.1. Top rewriting

The Maple™ statement `ruleName := [`l`,`r`]` declares the rewrite rule $l \to r$ as a pair and assigns it the name `ruleName`. The function `Transform` associates to any such pair the function applying the corresponding rewrite rule at the top of any term: Given a term t, the function application `Transform(ruleName)`(t) applies the rewrite rule $l \to r$ to t, as defined in Definition 1. If the rule cannot be applied i.e. if `t` does not match its left side l, then the exception `"Fail"` is raised. This is the standard *rewriting at the top* or *top rewriting* strategy.

Example 5. Consider the property $\int v + w \; dx = \int v \; dx + \int w \; dx$ of linearity of the integral. The rewrite rule corresponding to its application from left to right can be defined with symbtrans as the pair

```
IntegralLinearity := [
 Integral(A_ + B_, C_),
 Integral(A, C) + Integral(B, C)];
```

A convention in the package is that variable names end with "_" in order to distinguish them from constants. In order to apply the `IntegralLinearity` rule at the top to the term

```
t := Integral(v(x)+w(x),x);
```

we write `Transform(IntegralLinearity)(t)`. The resulting term is

```
Integral(v(x),x) + Integral(w(x),x)
```

3.2. Elementary transformations

The two elementary transformations `Identity` and `Fail` are defined as follows.

```
Identity(t) =def t
Fail(t)     =def error "Fail"
```

The first one has no effect, since it transforms `t` into itself. The second one always fails and raises the exception `"Fail"`. This exception is raised each time a transformation fails transforming a term. What is a failure for a transformation has to be defined for each transformation, as previously done for top rewriting.

It is sometimes useful to consider the non-progress of a transformation as a failure with the aim to handle this exception and enable other transformations. This feature is realized by the transformation combinator `IdentityAsFail` defined by

```
IdentityAsFail(s)(t) =def  if s(t) = t then Fail(t); else s(t);
```

for any transformation `s` and any term `t`.

Conversely, it is also convenient to hide at some higher level the exception `"Fail"` raised by a transformation `s`. This is the purpose of the `FailAsIdentity` combinator:

```
FailAsIdentity(s)(t) =def try s(t); catch "Fail" : t;
```

3.3. Transformation combinators

In this section we define three *transformation combinators*. They take transformations as parameters and control their order of application. Thus they help defining complex transformations by combination. They are generic in the sense that they do not depend on the nature and structure of the terms they are applied on. They are defined as follows.

`LeftChoice([s1, ..., sn])(t)`	$=_{def}$ `try s1(t); catch "Fail":LeftChoice([s2, ..., sn])(t)`
`LeftChoice([])(t)`	$=_{def}$ `Fail(t);`
`Comp([s1, ..., sn])(t)`	$=_{def}$ `Comp([s2, ..., sn])(s1(t));`
`Comp([])(t)`	$=_{def}$ `t;`
`STNormalizer(s)(t)`	$=_{def}$ `if s(t) = t then t; else STNormalizer(s)(s(t));`

The transformations `LeftChoice([`s`])` and `Comp([`s`])` are defined by induction on $n \geq 0$ for any sequence of transformations $s = (\mathtt{s}_i)_{i=1,\ldots,n}$. If $n = 0$, both do nothing. Otherwise, the application of `LeftChoice([`s`])` to the term `t` returns the result $\mathtt{s}_i$`(t)` of the application of the first transformation in the sequence s which succeeds on the term `t`. It reports a failure if no one succeeds. When $n \geq 1$ the transformation `Comp([`$\mathtt{s}_1$`, ..., `$\mathtt{s}_n$`])` applies $\mathtt{s}_1$, $\mathtt{s}_2$, etc in sequence, until application of $\mathtt{s}_n$ or a previous failure. The transformation `STNormalizer(s)` iterates the application of the transformation `s` until the latter fails or a fixed point is reached. The transformation `STNormalizer(s)` fails if and only if the transformation `s` fails during these iterated applications. To avoid the failure of the transformation `s` when computing the normal form of a term t with respect to `s`, one should write `STNormalizer(FailAsIdentity(s))(`t`)`. The transformations involving `STNormalizer` presents a risk of non termination and should therefore be carefully employed. Notice that the above definition of `STNormalizer` is just a specification. For more efficiency, the implementation computes `s(t)` only once.

3.4. Traversal transformations

This section introduces transformations called *traversal* or *term* transformations because they explore the structure of the term they are applied on. We provide three traversal transformation constructors: `Some`, `Outermost` and `Innermost`. By extension of a classical terminology in rewriting theory, a *redex* of a term t for a transformation s is a subterm of t that can be transformed by s, i.e. on which the transformation s does not fail.

The transformation `Some(s)` tries to apply the transformation `s` to all the immediate subterms of a term `t`. It fails if all these applications fail, or if `t` is a constant or a

variable. It is defined by

```
Some(s)(t) =def if t = f(t1,...,tn) then
                  if ∀i ∈ {1,...,n}  s(ti) fails, then
                    Fail(t);
                  else
                    f(FailAsIdentity(s)(t1),...,FailAsIdentity(s)(tn));
                  fi;
                else // t is a constant or a variable
                  Fail(t);
                fi;
```

Example 6. Consider again the rewrite rule `IntegralLinearity` of Example 5, encoding the linearity of the integral, and the term t encoding $\int v(x) + w(x)\ dx$. Notice that the statement

`Transform(IntegralLinearity)(`t`+2)`

raises the exception `"Fail"` because the rewrite rule cannot be applied at the top of the expression t+2. A solution is to replace it with the statement

`Some(Transform(IntegralLinearity))(`t`+2)`

which produces $\int v(x)\ dx + \int w(x)\ dx + 2$ since the expression t+2 is viewed as the term $+(t, 2)$.

The transformation `Some` is not very useful in practice. Its main purpose is to shorten the definition of the other traversal transformations, namely `Outermost` and `Innermost`.

The transformation `Outermost(s)` is very common in symbolic computation. It applies the transformation s once to all the redexes of t for s that are the closest ones to the root of t, i.e. to the largest subterms of t on which s succeeds. In other words the transformation `Outermost` traverses the term t down from its root and tries to apply s to each traversed subterm. If the transformation s succeeds on some subterm t' of t, then it is not applied to the proper subterms of t'. In particular, `Outermost`(s) fails if and only if s fails on all the subterms of t. It can be formally defined by

```
Outermost(s) =def LeftChoice([s,Some(Outermost(s))])
```

Example 7. Let

$$t = 2 + \underbrace{\int v(x) + 3(\underbrace{\int w(x) + g(x)\ dx}_{r_1})\ dx}_{r_0}$$

be a term with two redexes r_0 and r_1 for the rule of integral linearity. Then

`Outermost(IntegralLinearity)(`t`)`

gives the expression

$$2 + \int v(x)\ dx + \int 3(\int w(x) + g(x)\ dx)\ dx,$$

since the rule `IntegralLinearity` is only applied to the outermost redex r_0 of t.

The strategy `Innermost`(s) works similarly, but in the opposite direction, i.e. it traverses a term t up from its smallest subterms and tries to apply the strategy s once to the smallest redexes of t for s. It is formally defined by

```
Innermost(s) =def LeftChoice([Some(Innermost(s)),s])
```

Example 8. For the term t of Example 7, the expression
`Innermost(IntegralLinearity)`(t)
applies the rule `IntegralLinearity` only to the innermost redex r_1 of t and gives the expression

$$2 + \int v(x) + 3(\int w(x)\ dx + \int g(x)\ dx)\ dx.$$

We provide additional traversal transformations, namely `All`, `TopDown` and `BottomUp`. They are less useful in practice, but we include them in symbtrans because they exist in other strategy languages, e.g. the TOM strategy language (Balland et al., 2007).

The transformation `All(s)` applies the transformation `s` to all the immediate subterms of any term `t`, and it fails if and only if one of the applications to the immediate subterms fails. It is defined by

```
All(s)(t) =def if t = f(t1,...,tn) then
                  f(s(t1),...,s(tn));
               else // t is a constant or a variable
                  t;
               fi;
```

The main purpose of the transformation `All` is to simplify the definition of the transformations `TopDown` and `BottomUp`.

The transformation `TopDown(s)` tries to apply the transformation `s` to all the subterms of any term `t`, at any depth, by starting with the root of `t`. It fails when there is a subterm of `t` where `s` fails. It is defined by

```
TopDown(s) =def Comp([s,All(TopDown(s))])
```

The transformation `BottomUp(s)` behaves similarly, but works in the opposite direction, i.e. it starts from the leaves (the smallest subterms) and goes up.

```
BottomUp(s) =def Comp([All(BottomUp(s)),s])
```

4. Advanced features

In this section we introduce some technical features of symbtrans, namely, the ability to combine rewriting and procedural programming, the matching modulo associativity and commutativity and the conditional rewriting.

4.1. Procedural programming and strict evaluation

It is possible to combine rewrite rules and procedural programming in symbtrans. That is, the right-hand side of the rewrite rules may contain calls of Maple™ predefined functions or of functions defined by the user. This feature is not available in pure rewriting languages, e.g. in Maude (Clavel et al., 2007), but it is available in rewriting languages built upon a host language, e.g. ρ-log (Marin and Piroi, 2004) which is built on Mathematica and Tom (Balland et al., 2007) which is built on Java.

When we declare a rewrite rule whose right-hand side contains some function calls a problem occurs: since Maple™ is a strict evaluation language, it completely evaluates all the sub-expressions of an expression before evaluating the expression itself. Therefore it evaluates the function calls present in the right-hand side of the rewrite rules at the *declaration* of the rewrite rule, whereas the expected behaviour is most often evaluation of function calls at the *application* of the rewrite rule.

Example 9. Let us consider the following rule declaration:

```
e := [L_,nops(L)];
```

where `nops` is the Maple™ function that computes the number of arguments of `L`. When `L` is a list `nops(L)` computes the length of `L`. The problem is that Maple™ immediately evaluates `nops(L)` and the rule `e` becomes `[L_,1]`. The expected behavior is that the function `nops` is applied after the rule application, i.e. after instantiation of the variable `L_` by the substitution that arises from the matching of `L_` with a given term.

The solution we propose is to write `DelayEval ->` r instead of r for the right-hand side of a rewrite rule that contains function calls. In the example, the declaration of `e` would be

```
e := [L_, DelayEval -> nops(L)];
```

The Maple™ expression `x ->` r denotes the λ-term [3] $\lambda \mathtt{x}.r$, and this solution works for two reasons. First, the evaluation of r in $\lambda \mathtt{x}.r$ is delayed until an argument is provided to this λ-term. Second, Maple™ accepts the application of substitutions to λ-terms. With these two ingredients we can correctly implement the delayed evaluation in the rewrite rules as follows. The implementation of the application of the rule `[`l`, DelayEval ->` r`]` to the term t is done via the following steps:

(1) Compute a solution σ to the matching problem $l \ll t$,
(2) Compute the application $\sigma(r)$ of σ to r,
(3) Return the application `(DelayEval ->` $\sigma(r)$`)(DelayEval)` of the λ-term to the protected variable `DelayEval`.

[3] We recall that a λ-term is either a variable x, an abstraction $\lambda t.u$, or an application $(u\,v)$, where u and v are two λ-terms.

Since `DelayEval` is a protected variable in symbtrans and thus never appears in $\sigma(r)$, the expression `(DelayEval ->` $\sigma(r)$`)(DelayEval)` is β-reduced[4] by Maple™ to $\sigma(r)$.

It is worth mentioning that enclosing with unevaluation quotes `'...'` each function present in the right-hand side r of a rule, or using the parameter modifier `::uneval`, does not give the required solution. If an unevaluated function enclosed by the quotes appears as an argument of a term, then Maple™ eliminates the quotes and evaluates this function. As a consequence, applying a substitution to the term r provokes an early evaluation of the functions in r enclosed by quotes.

4.2. Extension to associative and commutative function symbols

When applying a rewrite rule $[l \to r]$ to a term t, and terms l and/or t contain associative and commutative function symbols, such as $+$ or $*$, then the matching problem $l \ll t$ and the rule application have to be done *modulo associativity and commutativity*.

The following definition generalizes Definition 1. It defines pattern-matching and rule application modulo a theory as in (Cirstea and Kirchner, 2001).

Definition 10. Let $\mathbb{T}$ be an equational theory. For two terms t and t' in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ the problem of finding substitutions σ such that the equality $\sigma(t') = t$ is a logical consequence of the axioms of $\mathbb{T}$ is called a *matching problem modulo* $\mathbb{T}$ and is denoted $t' \ll_{\mathbb{T}} t$. Such a substitution σ is called a *solution* of the matching problem $t' \ll_{\mathbb{T}} t$. The *application* of the rewrite rule $l \to r$ to a term t, denoted by $[l \to r](t)$, is the (possibly empty) set $\{\sigma_1(r), \sigma_2(r), \ldots\}$ where each σ_i is a solution of $l \ll_{\mathbb{T}} t$.

If the theory $\mathbb{T}$ axiomatizes the associativity and commutativity of some function symbols, then the set of solutions of the matching problem is finite. The associative and commutative symbols of symbtrans presently are $*$, $+$, $\cup$ and $\cap$. The corresponding theory is denoted by AC.

To deal with associative and commutative symbols the package symbtrans is equipped with two matching functions. The function `Matching(t')(t)` returns only one solution of the matching, always the same one, because transformations are expected to be reproducible. The function `MatchingAll(t')(t)` returns all the solutions. The first function is sufficient in many practical cases, but the second one is sometimes useful. In particular it is necessary for the application of conditional rules (see next section).

The present implementation of these two functions in symbtrans does not include optimizations suggested in the literature. In (Belkhir and Giorgetti, 2011), the authors suggested a more flexible approach. They provided a *lazy* matching algorithm modulo associativity and commutativity. *Lazy* means that the algorithm produces only a first solution and a way to get the other ones. We plan to integrate this lazy algorithm in a future version of symbtrans.

---

[4] The β-reduction is the reduction rule $(\lambda t.u)v \rightsquigarrow_\beta u[v/t]$, where $u[v/t]$ is the replacement of t with v in u.

4.3. Conditional rewriting

In symbtrans rewrite rules can be conditional. A conditional rule is a rewrite rule $l \to r$ and a condition c on the variables in l. In symbtrans a conditional rule is declared as a list `[l,r,c]` of three elements, where the third element is the condition. Notice that the `DelayEval` mechanism is usually required in the condition when it contains function calls.

Example 11. The linearity property (15) of an operator B can be encoded by the following conditional rewrite rule replacing $B(\alpha x)$ by $\alpha B(x)$ if α is a scalar:

```
[B(Alpha_*X_), Alpha*B(X), DelayEval -> IsScalar(Alpha)].
```

The application `Transform([l,r,c])(t)` of the conditional rule `[l,r,c]` to a term t, where c is a Boolean condition, gives a term $\sigma(r)$ where σ is a solution of the matching problem $l \ll_{AC} t$ such that $\sigma(c)$ holds, or raises the exception `"Fail"` if there is no such solution.

5. Formal proof examples

As typical examples, we consider the mathematical proofs reproduced in Appendices A and B. They are respectively formalized in Appendices C and D as sequences of transformations with the symbtrans language. Appendix E shows the collection of rules and transformations used in the proofs. The present section explains this formalization.

5.1. Mathematical operators

The algebraic properties of the integration, derivation, and general summation operators are encoded with symbtrans in Appendix E, Section E.1. Two versions of the Green rule are given in Appendix E, Section E.2: the usual rule `GreenRule`, and a parametrized conditional one `CondGreenRule`. The pattern `patt` is used to ensure that the Green rule is applied in the right way.

5.2. About linearity

We recall that an operator L is said to be linear if (15) and (16) hold. The linear operators used in the proofs are T, T^*, B, $\int_\Omega$, ∂_x and $\sum_i$. At the present level of formalization, scalars are not distinguishable from other symbolic expressions. As a consequence Eq. (15) cannot be turned into a general rewrite rule: otherwise this rule could also produce the unexpected term $v\ L(\alpha)$ from $L(\alpha\ v)$. We presently address this issue by writing one rewrite rule for each term α of interest in the proofs under consideration. It only concerns Steps 2 and 5 in Appendix B.

On the contrary Eq. (16) can be safely expressed by a rewrite rule. However the two-scale transform manipulates many linear operators and it is tedious to define a rewrite rule that expresses the linearity property (16) for each operator. Therefore we provide the generic constructor `Linearity(n,fun,t)`, where `fun` is a function and `t` is a term with `n` (underscored) variables or more. This constructor generates a rewrite rule that states that the operator `t` is linear with respect to its $\mathtt{n}^{\text{th}}$ variable, in the sense of `fun`. Notice that `fun` is usually the + function provided by Maple™. However it is possible to have different + functions for different vector spaces. For instance, the Maple™ expressions

```
Linearity(2,x->y->x+y,Integral(Omega_,_,Z_));
```

and

```
Linearity(1,x->y->x+y,T(_));
```

respectively express the linearity property (16) of the `Integral` and `T` operators with respect to +. Their evaluation respectively produces the rule

```
[Integral(Omega_,X_+Y_,Z_), Integral(Omega,X,Z)+Integral(Omega,Y,Z)]
```

and

```
[T(X_+Y_), T(X)+T(Y)].
```

a

For a given operator, the two rewrite rules that correspond to (15) and (16) can be merged together using transformation combinators. The collection of linear operators as well as their linearity properties is stored in an array named `LinearityOf` and defined in the file `integral.mpl` reproduced in Appendix E, Section E.1.

5.3. Convergence calculus

The notion of convergence has been introduced in Section 2. The theory of convergence (12) and (13) corresponds to the following rewrite rules:

$$O(\varepsilon) + O(\varepsilon) \to O(\varepsilon), \tag{17}$$

$$\sum_{i=1}^{n} O(\varepsilon) \to O(\varepsilon), \tag{18}$$

$$\int_{\Omega} O(\varepsilon)\, dx \to O(\varepsilon), \text{ and} \tag{19}$$

$$z * O(\varepsilon) \to O(\varepsilon) \tag{20}$$

for a term z bounded with respect to ε. These rules are combined in a strategy defined by

```
ConvergenceStrategy :=
  STNormalizer(
    LeftChoice([
      Outermost(OEpsilonSum),
      Outermost(OEpsilonSUM),
      Outermost(OEpsilonIntegral),
      Outermost(OEpsilonConst)
    ])
  );
```

where `OEpsilonSum`, `OEpsilonSUM`, `OEpsilonIntegral` and `OEpsilonConst` respectively encode the rewriting rules (17), (18), (19) and (20). They are defined in the file `convergence.mpl` reproduced in Appendix E, Section E.3. The result `ConvergenceStrategy` is a powerful strategy that reduces $O(\varepsilon)$ terms as much as possible. In the present case it can be shown that it always terminates and thus can be systematically applied after each tranformation step.

We now present and address a problem arising when embedding this strategy within a computer algebra system with strict evaluation. If the notion of a function that tends to 0 when ε tends to 0 is represented by the Maple™ expression $O(\varepsilon)$, then Maple™ simplifies any expression $O(\varepsilon) - O(\varepsilon)$ to zero, whereas $O(\varepsilon) - O(\varepsilon)$ should be simplified

to $O(\varepsilon)$. The solution we suggest consists in considering the term $O(i, \varepsilon)$ instead of $O(\varepsilon)$, where i is a *fresh* index. The term "fresh" means that the same index has never been produced before to construct such a term. This solution is natural because it basically relies on the mathematical semantics of the term $O(\varepsilon)$. That is, two occurrences of $O(\varepsilon)$ are two different functions, and it is natural to distinguish them using two different indexes. Technically, we provide a function `FreshIndex()` that returns a new index at each call. Moreover each occurrence of $O(\varepsilon)$ in the right-hand side of a rewrite rule is replaced by $O($`FreshIndex()`$, \varepsilon)$.

5.4. Two-scale calculus

The algebraic properties of the two-scale operators have been stated and mathematically proved in Appendix A of (Lenczner and Smith, 2007). They are formulated as rewriting rules in Appendix E, Section E.4.

The two-scale limit of the gradient operator corresponds to Eq. (74) of (Lenczner and Smith, 2007). This equality is proved with symbtrans in Appendix C. The mechanism of the formal proof is to reduce the left- and the right-hand sides of the equality A.7, and then to show that their reduced forms are equal up to $O(\varepsilon)$.

The formal derivation of the two-scale model of the stationary heat equation with symbtrans is reproduced in Appendix D. There, the transformation steps are applied to the term `LHST` − `RHST`, where `LHST` (resp. `RHST`) is the left- (resp. right-) hand side term of the heat equation (1). The two-scale model derivation uses the weak convergence property of the derivative operator (Step 6 in Appendix B) as a lemma. In the formal proof this lemma is written "by hand" as a rewriting rule after being formally proved.

5.5. Final remarks

Some proof steps require to apply some equation $A = B$ from left to right and from right to left. However, including the two rewriting rules $A \to B$ and $B \to A$ in the same strategy may induce non-termination. This is typically the case for the linearity properties. In this case, non-termination is avoided by the introduction of a more specialized version of the rules corresponding to the second orientation, such as:

```
AdhocSimplify := [Integral(Omega_,C_*SUM(F_,J_,D_),X_),
                  SUM(Integral(Omega,C*F,X),J,D)];
```

Notice that the proofs use two Maple™ functions: `has(expr1,expr2)` that returns true if `expr2` is a subexpression of `expr1`, and `Evala(Expand(expr))` that expands products and powers of rational functions with algebraic coefficients.

We have also developed a formal proof of the property (7) that the two-scale weak limit u^0 of Tu^ε is independent of y. This proof is not reproduced here. Instead, (7) is encoded as a lemma (See Appendix E.5) used in the main proofs. The hypotheses of these proofs are gathered in the file `hypothesis.mpl` reproduced in Appendix E, Section E.6.

6. Theoretical basis and related work

The theoretical basis for this work is the deterministic fragment of the ρ_{AC}-calculus (Cirstea and Kirchner, 2001), where AC is the theory axiomatizing the associativity and commutativity of the symbols $+, *, \cup$ and $\cap$. The $\rho_{\mathbb{T}}$-calculus, where $\mathbb{T}$ is an equational

theory, is an extension of the λ-calculus; one abstracts on a pattern rather than on a single variable. The abstraction mechanism is based on the rewrite rule $l \to r$, also viewed as a ρ-term. Notice that when l is just a variable x, this ρ-term corresponds to the λ-term $\lambda x.\ r$. Moreover, the $\rho_{\mathbb{T}}$-calculus considers higher-order terms, i.e. terms that may contain abstractions and rule applications.

When an abstraction $l \to r$ is applied to a ρ-term t, which is denoted by $[l \to r]t$, the matching mechanism is based on the binding of the free variables of l to the appropriate subterms of t. This matching is done modulo the theory $\mathbb{T}$. The latter is often expressed by algebraic axioms such as associativity and/or commutativity. In symbtrans both the left-hand side term of a rule and the term under rule application corresponds to first-order ρ-terms, i.e. they contain neither abstractions nor rule applications. However the right-hand side of a rule may be a higher-order ρ-term, since it may contain function calls. Those functions are nothing but λ-terms. Handling the priority between rule application and β-reduction is explained in the steps (1), (2) and (3) in Section 4.1. Despite the fact that strategies can be encoded with the $\rho_{\mathbb{T}}$-calculus (Cirstea et al., 2003) by means of some constructors, we preferred encoding the strategies in symbtrans by means of Maple functions for the sake of efficiency. Finally, we notice that in the symbtrans language, if a rule cannot be applied to a term then the exception `Fail` is raised. This makes a subtle difference with the semantics of the $\rho_{\mathbb{T}}$-calculus that consists in returning an empty set in this case. The problem of the $\rho_{\mathbb{T}}$-calculus approach is that we can not distinguish between an empty set which is a mathematical term that could arise from the symbolic transformations, and the empty set which denotes the failure of the application of a rule.

The proposed transformation language does not claim for originality. It is deliberately an adaptation for Maple of popular strategy languages such as ρ-log (Marin and Piroi, 2004) or Tom (Balland et al., 2007). But, departing from Tom which extends an host language with an additive syntax, our transformation language smoothly integrates with standard Maple functions. Consequently, the Maple programmer learns it quickly, and is free to mix function- and rule-based programming styles. Moreover all the features of her development environment (such as refactoring, code completion, dependency analyses, etc) are preserved for free.

The closest implementation is ρ-log, a package developed upon the advanced rewriting kernel of Mathematica. It supports non-deterministic and conditional rewriting. The main drawback of ρ-log is that it considers the non-applicability of a rule as the identity. Technically speaking, the strategy `FailAsIdentity` is implicitly applied to all the transformations. However, when a transformation returns the same term given as an input, we do not know if this transformation fails or it performs some modifications and then returns the same term. Moreover in ρ-log it is not possible, at least in a straightforward way, to do higher-order rewriting, since the rewriting rules are not directly accessible to the user: They are declared by means of the constructor `DeclareRule`.

7. Conclusion

Our main motivation for the development of a transformation language in Maple™ was to facilitate the design of the MEMSALab software dedicated to the automatic derivation of multi-scale models. However symbtrans is a general tool that can be used by Maple™ programmers and mathematicians in the formalization of equational reasoning. It makes it possible to express rule-based symbolic computations in a concise and natural

way, thus providing a good guarantee of the correctness of the formal proofs with respect to their hand-written counterparts. Since the symbtrans package is written in Maple™, it obviously does not extend the expressivity of the Maple™ language, but it clearly increases readability and conciseness. Although this paper presents an implementation in Maple™™, the transformations presented here could easily be developed in a similar way in any functional language.

The transformation language symbtrans allows the derivation of the weak two-scale limit of the derivative operator and the two-scale model of the stationary heat equation at the same "level" as the mathematical derivations. The word "level" covers three aspects: the formal and the hand-written proofs have almost the same size, they follow the same steps, and the strategy term written at each step of the formal proofs is a natural formalization of its mathematical counterpart. The symbtrans package is used in (Yang et al., 2011a,b) to formally derive the two-scale model of the stationary heat equation in a region composed of a thin part and a part with periodically distributed holes.

For a more scalable treatment of linearity we plan in a near future to detect the scalar nature of terms by assigning a type to each expression. More generally a type system for mathematical expressions is under way. We plan to transform each proof into a module whose execution produces a parametrized rewrite rule. The latter can be instantiated and applied in other proofs.

Acknowledgements

The authors would like to thank the anonymous reviewers of a previous version of this paper for their helpful comments, and R. N. Dhara and B. Yang for the feedback provided by their development activity using symbtrans.

References

Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A., 2007. Tom: Piggybacking rewriting on Java. In: the proceedings of the 18th International Conference on Rewriting Techniques and Applications RTA 07. pp. 36–47.

Belkhir, W., Giorgetti, A., 2011. Lazy AC-pattern matching for rewriting. In: Escobar, S. (Ed.), Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011. Vol. 82 of EPTCS. pp. 37–51.

Bensoussan, A., Lions, J.-L., Papanicolaou, G., 1978. Asymptotic analysis for periodic structures. Vol. 5 of Studies in Mathematics and its Applications. North-Holland Publishing Co., Amsterdam.

Borovansky, P., Kirchner, C., Kirchner, H., Ringeissen, C., 2001. Rewriting with strategies in ELAN: a functional semantics. International Journal of Foundations of Computer Science 12 (01), 69–95.

Bündgen, R., 1995. Combining computer algebra and rule based reasoning. In: Proceedings of The International Conference on Integrating Symbolic Mathematical Computation and Artificial Intelligence. Vol. 958 of LNCS. Springer, pp. 209–223.

Casado-Díaz, J., 2000. Two-scale convergence for nonlinear Dirichlet problems in perforated domains. Proc. Roy. Soc. Edinburgh Sect. A 130 (2), 249–276.

Ciarlet, P. G., 1988. Mathematical elasticity. Vol. I. Vol. 20 of Studies in Mathematics and its Applications. North-Holland Publishing Co., Amsterdam.

Cioranescu, D., Damlamian, A., Griso, G., 2002. Periodic unfolding and homogenization. C. R. Math. Acad. Sci. Paris 335 (1), 99–104.

Cioranescu, D., Damlamian, A., Griso, G., 2008. The periodic unfolding method in homogenization. SIAM Journal on Mathematical Analysis 40 (4), 1585–1620.

Cirstea, H., Kirchner, C., May 2001. The rewriting calculus — Part I and II. Logic Journal of the Interest Group in Pure and Applied Logics 9 (3), 427–498.

Cirstea, H., Kirchner, C., Liquori, L., 2001. The rho cube. In: Honsell, F., Miculan, M. (Eds.), Foundations of Software Science and Computation Structures. Vol. 2030 of LNCS. Springer, pp. 168–183.
URL http://dx.doi.org/10.1007/3-540-45315-6_11

Cirstea, H., Kirchner, C., Liquori, L., Wack, B., 2003. Rewrite strategies in the rewriting calculus. In: Gramlich, B., Lucas, S. (Eds.), 3rd International Workshop on Reduction Strategies in Rewriting and Programming . Vol. 86(4) of Electronic Notes in Theoretical Computer Science. Elsevier, Valencia, Spain, pp. 18–34.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. L. (Eds.), 2007. All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic. Vol. 4350 of LNCS. Springer.

Dautray, R., Lions, J.-L., 1990. Mathematical analysis and numerical methods for science and technology. Vol. 3. Springer, Berlin.

Fèvre, S., Wang, D., 1998. Combining algebraic computing and term-rewriting for geometry theorem proving. In: Calmet, J., Plaza, J. (Eds.), Artificial Intelligence and Symbolic Computation. Vol. 1476 of LNCS. Springer, pp. 145–156.
URL http://dx.doi.org/10.1007/BFb0055909

Geuvers, H., 2009. Introduction to type theory. In: Bove, A., Barbosa, L., Pardo, A., Pinto, J. (Eds.), Language Engineering and Rigorous Software Development. Vol. 5520 of LNCS. Springer, pp. 1–56.

Lenczner, M., 1997. Homogénéisation d'un circuit électrique. C. R. Acad. Sci. Paris Sér. II b 324 (9), 537–542.

Lenczner, M., Senouci-Bereksi, G., 1999. Homogenization of electrical networks including voltage-to-voltage amplifiers. Math. Models Methods Appl. Sci. 9 (6), 899–932.

Lenczner, M., Smith, R. C., 2007. A two-scale model for an array of AFM's cantilever in the static case. Mathematical and Computer Modelling 46 (5-6), 776–805.

Marin, M., Piroi, F., 2004. Rule-based programming with mathematica. In: In Sixth Mathematica Symposium (IMS 2004). pp. 1–6.

Yang, B., Belkhir, W., Dhara, R. N., Lenczner, M., Giorgetti, A., Apr. 2011a. Computer–aided multiscale model derivation for MEMS arrays. In: EuroSimE 2011, 13-th Int. Conf. on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems. IEEE Computer Society, Linz, Austria, 6 pages. Electronic proceedings.

Yang, B., Dhara, R. N., Belkhir, W., Lenczner, M., Giorgetti, A., Aug. 2011b. Formal methods for multiscale models derivation. In: CFM 2011, 20-th Congrès Français de Mécanique. Besançon, France, 5 pages. Electronic proceedings.

A. A mathematical proof of the derivative weak convergence property

This section is devoted to the proof of Proposition 3. We start with some reminders of mathematics that complement those in Section 2.2.

Here the two-scale transform T can be viewed as a linear continuous operator from $L^2(\Omega)$ into $L^2(\widetilde{\Omega}\times Y)$, as such its adjoint T^* is a linear continuous operator from $L^2(\widetilde{\Omega}\times Y)$ into $L^2(\Omega)$ defined by

$$\int_{\widetilde{\Omega}\times Y} T(u)\ v\ dxdy = \int_{\Omega} u\ T^*(v)\ dx \text{ for any } u \in L^2(\Omega) \text{ and } v \in L^2(\widetilde{\Omega}\times Y). \tag{A.1}$$

The two-scale transform can also be defined on integrable functions and it satisfies the property

$$T(u\ v) = T(u)T(v) \text{ for any } u, v \in L^2(\Omega). \tag{A.2}$$

Then, we define the so-called *regularized inverse two-scale transform* $B : L^2(\widetilde{\Omega}\times Y) \to L^2(\Omega)$ by

$$B(v)(x) = v(x, x/\varepsilon).$$

It can be easily checked that B is a linear operator. The partial derivatives of $B(v)$ for any sufficiently regular function v can be derived by applying the chain rule

$$\partial_{x_i} B(v) = B(\partial_{x_i} v) + \frac{1}{\varepsilon} B(\partial_{y_i} v). \tag{A.3}$$

It is also useful to know how the null condition of a function $v(x, y)$ on the boundary $\partial(\widetilde{\Omega}\times Y)$ is transfered to its range by B:

$$\text{If } v = 0 \text{ on } \partial(\widetilde{\Omega}\times Y) \text{ then } B(v) = 0 \text{ on } \partial\Omega. \tag{A.4}$$

In the following lemma (admitted), $O(\varepsilon)$ denotes any function that vanishes in the $L^2(\Omega)$-norm when ε tends to zero.

Lemma 12. *The operator B is a zero-order approximation of the adjoint operator T^* in the sense that*

$$T^*(v) - B(v) = O(\varepsilon) \tag{A.5}$$

for any sufficiently regular and Y-periodic function v. Moreover, $B(v)$ can be approximated at the first-order by

$$B(v) = T^*(v + \varepsilon \sum_{j=1}^{n} y_j \partial_{x_j} v) + \varepsilon O(\varepsilon). \tag{A.6}$$

In the following, for simplicity we write u instead of u^ε. We shall prove Proposition 3 or equivalently, by the density of the set $\mathcal{C}_0^\infty(\widetilde{\Omega}\times Y)^n$ of infinitely continuously differentiable functions will all derivatives vanishing on $\partial(\widetilde{\Omega}\times Y)$ in the set $L^2(\widetilde{\Omega}\times Y)^n$, that

$$\underbrace{\sum_{i=1}^{n} \int_{\widetilde{\Omega}\times Y} T(\partial_{x_i} u) v_i\ dxdy}_{\Psi} = \sum_{i=1}^{n} \int_{\widetilde{\Omega}\times Y} (\partial_{x_i} u^0 + \partial_{y_i} u^1) v_i\ dxdy + O(\varepsilon) \tag{A.7}$$

for any $v = (v_1, ..., v_n) \in \mathcal{C}_0^\infty(\widetilde{\Omega}\times Y)^n$.

- **Step 1.** Applying the definition of T^* to the left-hand side Ψ of (A.7) yields

$$\Psi = \sum_{i=1}^{n} \int_{\Omega} \partial_{x_i} u \, T^*(v_i) \, dx.$$

- **Step 2.** From the approximation (A.5) of $T^*(v_i)$ by $B(v_i)$, the linearity of integral, the boundedness of $||\partial_{x_i} u||_{L^2(\Omega)}$ and the property (13) of $O(\varepsilon)$ we get

$$\begin{aligned}\Psi &= \sum_{i=1}^{n} \int_{\Omega} \partial_{x_i} u \, B(v_i) \, dx + \sum_{i=1}^{n} \int_{\Omega} \partial_{x_i} u \, O(\varepsilon) \, dx \\ &= \sum_{i=1}^{n} \int_{\Omega} \partial_{x_i} u \, B(v_i) \, dx + O(\varepsilon).\end{aligned}$$

- **Step 3.** Then, we apply the Green formula (14) and get

$$\Psi = \sum_{i=1}^{n} \int_{\partial\Omega} u \; B(v_i)(n_x)_i \; ds(x) - \sum_{i=1}^{n} \int_{\Omega} u \; \partial_{x_i} B(v_i) \; dx + O(\varepsilon),$$

  and the terms on the boundary are removed thanks to Property (A.4),

$$\Psi = -\sum_{i=1}^{n} \int_{\Omega} u \; \partial_{x_i} B(v_i) \; dx + O(\varepsilon).$$

- **Step 4.** From the expression (A.3) applied to the partial derivatives of $B(v_i)$ and by linearity of integral,

$$\Psi = -\sum_{i=1}^{n} \int_{\Omega} u \, \left(B(\partial_{x_i} v_i) + \frac{1}{\varepsilon} B(\partial_{y_i} v_i) \right) \, dx + O(\varepsilon),$$

  and

$$\Psi = -\sum_{i=1}^{n} \big[\underbrace{\int_{\Omega} u B(\partial_{x_i} v_i) \; dx}_{\Psi_1} + \underbrace{\int_{\Omega} \frac{1}{\varepsilon} u B(\partial_{y_i} v_i) \; dx}_{\Psi_2} \big] + O(\varepsilon).$$

- **Step 5.** We apply (A.5) and (A.6) to approximate $B(\partial_{x_i} v_i)$ at the zero-order and $B(\partial_{y_i} v_i)$ at the first-order together with the rule (13) and thus get

$$\Psi_1 = \int_{\Omega} u \; T^*(\partial_{x_i} v_i) \, dx + O(\varepsilon)$$

  and

$$\Psi_2 = \int_{\Omega} \frac{1}{\varepsilon} u \, \big[T^*(\partial_{y_i} v_i + \varepsilon \sum_{j=1}^{n} y_j \partial_{x_j} \partial_{y_i} v_i) \big] dx + O(\varepsilon).$$

  Thanks to the linearity of T^*,

$$\Psi_2 = \int_{\Omega} u \; T^*(\frac{1}{\varepsilon} \partial_{y_i} v_i + \sum_{j=1}^{n} y_j \partial_{x_j} \partial_{y_i} v_i) \; dx + O(\varepsilon).$$

Grouping Ψ_1 and Ψ_2,

$$\Psi = -\sum_{i=1}^{n} \Big[\int_{\Omega} u\, T^*(\partial_{x_i} v_i)\, dx + \int_{\Omega} u\, T^*(\frac{1}{\varepsilon}\partial_{y_i} v_i + \sum_{j=1}^{n} y_j \partial_{x_j}\partial_{y_i} v_i)\; dx \Big] + O(\varepsilon).$$

- **Step 6.** From the definition of the dual operator T^* of T,

$$\Psi = -\sum_{i=1}^{n} \Big[\int_{\widetilde{\Omega}\times Y} T(u)\; (\partial_{x_i} v_i)\, dxdy + \int_{\widetilde{\Omega}\times Y} T(u)\; (\frac{1}{\varepsilon}\partial_{y_i} v_i + \sum_{j=1}^{n} y_j \partial_{x_j}\partial_{y_i} v_i)\; dxdy \Big] + O(\varepsilon).$$

After expanding and applying linearity of integral,

$$\begin{aligned}\Psi = -\sum_{i=1}^{n} \Big[& \int_{\widetilde{\Omega}\times Y} T(u)\; (\partial_{x_i} v_i)\, dxdy + \int_{\widetilde{\Omega}\times Y} T(u) \frac{1}{\varepsilon}\partial_{y_i} v_i\, dxdy \\ & + \int_{\widetilde{\Omega}\times Y} T(u) \sum_{j=1}^{n} y_j \partial_{x_j}\partial_{y_i} v_i\; dxdy \Big] + O(\varepsilon).\end{aligned}$$

- **Step 7.** We use the zero-order approximation

$$Tu^{\varepsilon} = u^0 + \varepsilon O(\varepsilon), \tag{A.8}$$

and the first-order approximation, both deduced from (10), respectively in the first and third integrals and in the second integral to get

$$\begin{aligned}\Psi = -\sum_{i=1}^{n} \Big[& \int_{\widetilde{\Omega}\times Y} (u^0 + O(\varepsilon))\; \partial_{x_i} v_i\; dxdy \\ & + \int_{\widetilde{\Omega}\times Y} \big(u^0 + \varepsilon u^1 + \varepsilon \sum_{j=1}^{n} y_j \partial_{x_j} u^0 + \varepsilon O(\varepsilon)\big) \frac{1}{\varepsilon}\partial_{y_i} v_i\; dxdy \\ & + \int_{\widetilde{\Omega}\times Y} (u^0 + O(\varepsilon))\; \sum_{j=1}^{n} y_j \partial_{x_j}\partial_{y_i} v_i\; dxdy \Big] + O(\varepsilon).\end{aligned}$$

After simplification

$$\begin{aligned}\Psi = -\sum_{i=1}^{n} \Big[& \int_{\widetilde{\Omega}\times Y} u^0\; \partial_{x_i} v_i\; dxdy + \int_{\widetilde{\Omega}\times Y} \frac{1}{\varepsilon} u^0 \partial_{y_i} v_i\; dxdy \\ & + \int_{\widetilde{\Omega}\times Y} u^1 \partial_{y_i} v_i\; dxdy + \sum_{j=1}^{n} \int_{\widetilde{\Omega}\times Y} y_j \partial_{x_j} u^0\; \partial_{y_i} v_i\; dxdy \\ & + \int_{\widetilde{\Omega}\times Y} u^0 \sum_{j=1}^{n} y_j \partial_{x_j}\partial_{y_i} v_i\; dxdy \Big] + O(\varepsilon).\end{aligned}$$

- **Step 8.** We apply the following instance of the Green formula to the second subterm,

$$\int_{Y} u^0 \partial_{y_i} v_i\; dy = \int_{\partial Y} u^0 v_i n_{y_i}\; ds(y) - \int_{Y} \partial_{y_i} u^0\; v_i\; dy,$$

where n_y stands for the unit outward normal vector to the boundary ∂Y of Y. Remarking that $\partial_{y_i} u^0$ vanishes [5] and that v vanishes on ∂Y, the second subterm vanishes and we obtain

$$\Psi = -\sum_{i=1}^{n} \Big[\int_{\widetilde{\Omega}\times Y} u^0 \, \partial_{x_i} v_i \, dxdy + \int_{\widetilde{\Omega}\times Y} u^1 \partial_{y_i} v_i \, dxdy + \int_{\widetilde{\Omega}\times Y} \sum_{j=1}^{n} y_j \partial_{x_j} u^0 \, \partial_{y_i} v_i \, dxdy + \int_{\widetilde{\Omega}\times Y} u^0 \sum_{j=1}^{n} y_j \, \partial_{x_j} \partial_{y_i} v_i \, dxdy \Big] + O(\varepsilon).$$

- **Step 9.** Similarly, repeating the Green formula application until no derivative is left on the test function v,

$$\begin{aligned}
\Psi = -\sum_{i=1}^{n} \Big[& \int_{\partial\widetilde{\Omega}\times Y} u^0 \; v_i (n_x)_i \, ds(x)dy - \int_{\widetilde{\Omega}\times Y} \partial_{x_i} u^0 \, v_i \, dxdy \\
& + \int_{\widetilde{\Omega}\times\partial Y} u^1 v_i n_{y_i} \, dxds(y) - \int_{\widetilde{\Omega}\times Y} \partial_{y_i} u^1 \, v_i \, dxdy \\
& + \sum_{j=1}^{n} \int_{\widetilde{\Omega}\times\partial Y} y_j \, \partial_{x_j} u^0 \, v_i n_{y_i} \, dxds(y) - \sum_{j=1}^{n} \int_{\widetilde{\Omega}\times Y} \partial_{y_i}(y_j \partial_{x_j} u^0) \, v_i \, dxdy \\
& + \sum_{j=1}^{n} \int_{\partial\widetilde{\Omega}\times\partial Y} u^0 v_i n_{y_i} \, y_j n_{x_j} \, ds(x)ds(y) \\
& - \sum_{j=1}^{n} \int_{\partial\widetilde{\Omega}\times Y} \partial_{y_i}(y_j u^0) \, n_{x_j} v_i \, ds(x)dy \\
& - \sum_{j=1}^{n} \int_{\widetilde{\Omega}\times\partial Y} y_j \partial_{x_j} u^0 \; v_i n_{y_i} \, dxds(y) \\
& + \sum_{j=1}^{n} \int_{\widetilde{\Omega}\times Y} \partial_{y_i}(y_j \partial_{x_j} u^0) \; v_i \, dxdy \Big] + O(\varepsilon).
\end{aligned}$$

Since v vanishes on all boundaries,

$$\begin{aligned}
\Psi = \sum_{i=1}^{n} \Big[& \int_{\widetilde{\Omega}\times Y} \partial_{x_i} u^0 v_i \, dxdy + \int_{\widetilde{\Omega}\times Y} \partial_{y_i} u^1 v_i \, dxdy \\
& + \sum_{j=1}^{n} \int_{\widetilde{\Omega}\times Y} \partial_{y_i}(y_j \partial_{x_j} u^0) \, v_i \, dxdy \\
& - \sum_{j=1}^{n} \int_{\widetilde{\Omega}\times Y} \partial_{y_i}(y_j \partial_{x_j} u^0) \, v_i \, dxdy \Big] + O(\varepsilon),
\end{aligned}$$

[5] In the formal proof this simplification is done after the successive applications of Green rule and the elimination of the boundary terms, i.e. at the end of step 9.

or after simplification,

$$\Psi = \sum_{i=1}^{n} \Big[\int_{\widetilde{\Omega}\times Y} \partial_{x_i} u^0 v_i \; dxdy + \int_{\widetilde{\Omega}\times Y} \partial_{y_i} u^1 v_i \; dxdy \Big] + O(\varepsilon).$$

Finally, thanks to linearity of integral and by factoring v_i,

$$\Psi = \sum_{i=1}^{n} \Big[\int_{\widetilde{\Omega}\times Y} (\partial_{x_i} u^0 + \partial_{y_i} u^1) v_i \; dxdy \Big] + O(\varepsilon).$$

B. A mathematical two-scale transformation of the heat equation

We detail the proof of Proposition 4. We start with test functions $v^0 \in \mathcal{C}_0^\infty(\widetilde{\Omega})$ and $v^1 \in \mathcal{C}^\infty(\widetilde{\Omega} \times Y)$ that is Y-periodic.

- **Step 1.** We choose $v = B(v^0 + \varepsilon v^1)$ as a test function in the weak formulation (1) of the model,

$$\sum_{i=1}^{n} \int_\Omega a(\partial_{x_i} u)\, \partial_{x_i} B(v^0 + \varepsilon v^1) \; dx = \int_\Omega f\, B(v^0 + \epsilon v^1) \; dx.$$

- **Step 2.** Applying the rule (A.3) of partial derivatives of $B(v)$ yields

$$\sum_{i=1}^{n} \int_\Omega a(\partial_{x_i} u) \; (B(\partial_{x_i}(v^0 + \varepsilon v^1)) + \frac{1}{\varepsilon} B(\partial_{y_i}(v^0 + \varepsilon v^1)) \; dx = \int_\Omega f\, B(v^0 + \epsilon v^1) \; dx.$$

By linearity of ∂ and B, and since v^0 does not depend on y, we get after application of (13) and simplifications,

$$\sum_{i=1}^{n} \int_\Omega a(\partial_{x_i} u) \; (B(\partial_{x_i} v^0) + B(\partial_{y_i} v^1)) \; dx = \int_\Omega f\, B(v^0) \; dx + O(\varepsilon).$$

From the linearity of B again,

$$\sum_{i=1}^{n} \int_\Omega a(\partial_{x_i} u) \; B(\partial_{x_i} v^0 + \partial_{y_i} v^1) \; dx = \int_\Omega f\, B(v^0) \; dx + O(\varepsilon).$$

- **Step 3.** The zero-order approximation (A.5) of B by T^* implies

$$\sum_{i=1}^{n} \int_\Omega a(\partial_{x_i} u) \; T^*(\partial_{x_i} v^0 + \partial_{y_i} v^1) \; dx = \int_\Omega f\, T^*(v^0) \; dx + O(\varepsilon).$$

- **Step 4.** Now, we apply the definition of the adjoint T^* of T,

$$\sum_{i=1}^{n} \int_{\widetilde{\Omega}\times Y} T(a\partial_{x_i} u) \; (\partial_{x_i} v^0 + \partial_{y_i} v^1) \; dxdy = \int_{\widetilde{\Omega}\times Y} T(f)\, v^0 \; dxdy + O(\varepsilon).$$

- **Step 5.** From the identity (A.2) and the assumptions (11) we get

$$\sum_{i=1}^{n} \int_{\widetilde{\Omega}\times Y} a^0\, T(\partial_{x_i} u) \; (\partial_{x_i} v^0 + \partial_{y_i} v^1) \; dxdy = \int_{\widetilde{\Omega}\times Y} f^0\, v^0 \; dxdy + O(\varepsilon).$$

- **Step 6.** From the approximation (A.7) of the derivative operator applied to the test function $a^0(\partial_{x_i} v^0 + \partial_{y_i} v^1)$ we get the wanted two-scale model

$$\sum_{i=1}^{n} \int_{\widetilde{\Omega}\times Y} a^0 \left(\partial_{x_i} u^0 + \partial_{y_i} u^1\right) \left(\partial_{x_i} v^0 + \partial_{y_i} v^1\right) dxdy = \int_{\widetilde{\Omega}\times Y} f^0 \, v^0 \, dxdy + O(\varepsilon).$$

C. A formal proof of the derivative weak convergence property

```
# File gradient.mpl
# Contributors: Walid Belkhir, Alain Giorgetti and Michel Lenczner

# 1. Library loading

new_lib_dir := "../lib":
libname := new_lib_dir, libname:
with(stmodule);

# 2. Rewrite systems loading

read ‘integral.mpl‘:
read ‘Green.mpl‘:
read ‘twoscale.mpl‘:
read ‘convergence.mpl‘:
read ‘lemmas.mpl‘:
read ‘hypothesis.mpl‘:

# 3. Proof

# Initial term
LHSterm := SUM(
  Integral([TSTM(Omega),TSTm(Omega)],T(partial(u,x(i))) * v(i),[x,y]),
  i,Iset);

# Step 1
LHSterm := Outermost(TwoScaleAdjointInv)(LHSterm);

# Step 2
LHSterm := Outermost(ApproximationTS[1])(LHSterm);
LHSterm := evala(Expand(LHSterm));
LHSterm := IntegrationStrategy(LHSterm);
LHSterm := ConvergenceStrategy(LHSterm);

# Step 3
LHSterm := Outermost(GreenRule)(LHSterm);
LHSterm := OutermostNF(IntegralOnBoundary(v(i))(partialBound))(LHSterm);
LHSterm := IntegrationStrategy(LHSterm);

# Step 4
LHSterm := Outermost(PartialOfB)(LHSterm);
LHSterm := evala(Expand(LHSterm));
```

```
LHSterm := IntegrationStrategy(LHSterm);

# Step 5
ApproximationTSInv2Context :=
 Comp([
  [(1/Epsilon)*X_,(1/Epsilon)*X],
  Outermost(ApproximationB[2](x(j))(y(j))(j)) ]);
LHSterm := OutermostNF(ApproximationTSInv2Context)(LHSterm);
LHSterm := Outermost(ApproximationB[1])(LHSterm);
LHSterm := evala(Expand(LHSterm));
LHSterm := Outermost(LinearityOf[TS])(LHSterm);
LHSterm := evala(Expand(LHSterm));
LHSterm := Outermost(LinearityBasic[TS])(LHSterm);
LHSterm := evala(Expand(LHSterm));
LHSterm := IntegrationStrategy(LHSterm);
LHSterm := ConvergenceStrategy(LHSterm);

# Step 6
LHSterm := Outermost(TwoScaleAdjoint)(LHSterm);
LHSterm := evala(Expand(LHSterm));
LHSterm := IntegrationStrategy(LHSterm);

# Step 7
ApproximationT2Context :=
 Comp([
  [(1/Epsilon)*X_,(1/Epsilon)*X],
  Outermost(ApproximationT[2])
 ]);
LHSterm := Outermost(ApproximationT2Context)(LHSterm);
LHSterm := Outermost(ApproximationT[1])(LHSterm);
LHSterm := evala(Expand(LHSterm));
LHSterm := IntegrationStrategy(LHSterm);
LHSterm := ConvergenceStrategy(LHSterm);
AdhocSimplify := [
  Integral(Omega_,C_*SUM(F_,J_,D_),X_),
  SUM(Integral(Omega,C*F,X),J,D)];
LHSterm := Outermost(AdhocSimplify)(LHSterm);

# Steps 8 and 9 of the mathematical proof.
# Differ from the mathematical proof: All the Green rule are performed.

LHSterm := STNormalizer(
 OutermostNF(Comp([
    OutermostNF(CondGreenRule(v(i))),
    Comp([OutermostNF(IntegralOnBoundary(v(i))(partialBound)),IntegrationStrategy])
  ])
 )
)(LHSterm);

LHSterm := OutermostNF(U0IndependentOfY)(LHSterm);
LHSterm := IntegrationStrategy(LHSterm);
```

```
LHSterm := OutermostNF(IndependentOfX)(LHSterm);

# Right-hand side term

RHSterm :=
 SUM(
   Integral(
     [TSTM(Omega),TSTm(Omega)],
     (partial(u0,x(i)) + partial(u1,y(i)))* v(i),
     [x,y]),
   i,Iset)
 + BigO(FreshIndex(),Epsilon);
RHSterm := evala(Expand(RHSterm));
RHSterm := IntegrationStrategy(RHSterm);

# ---- The left- and right-hand side terms match modulo O(epsilon).

Result := ConvergenceStrategy(LHSterm-RHSterm);
quit;
```

D. A formal two-scale transformation of the heat equation

```
# File heat.mpl
# Contributors: Walid Belkhir, Alain Giorgetti and Michel Lenczner

# 1. Library loading

new_lib_dir := "../lib":
libname := new_lib_dir, libname:
with(stmodule);

# 2. Rewrite systems loading

read ‘integral.mpl‘:
read ‘Green.mpl‘:
read ‘twoscale.mpl‘:
read ‘convergence.mpl‘:
read ‘lemmas.mpl‘:
read ‘hypothesis.mpl‘:

# 3. Lemmas

GradientApprox := [
  SUM(
    Integral([TSTM(Omega), TSTm(Omega)],a0*T(partial(U_,X1_))*V_,[X2_, Y_]),
    Iset_,K_),
  DelayEval -> SUM(
    Integral([TSTM(Omega), TSTm(Omega)],
      a0*(partial(u0,X1)+partial(u1,Y(Iset)))*V,[X2, Y]),
    Iset,K)+BigO(FreshIndex(),Epsilon)];
```

```
# 4. Proof

# Initial term. The equality is replaced by a difference
# to get more simplifications.
EqTerm := SUM(Integral(Omega,a * partial(u,x(i)) * partial(v,x(i)),x),i,Iset)
 - Integral(Omega,f*v,x);

# Step 1
TestFunChoice[1] := [v,B(v0+Epsilon*v1)];
EqTerm := Outermost(TestFunChoice[1])(EqTerm);

# Step 2
EqTerm := Outermost(PartialOfB)(EqTerm);
EqTerm := Outermost(LinearityOf[partial])(EqTerm);
EqTerm := Outermost(Linearity2Of[partial])(EqTerm);
EqTerm := Outermost(LinearityOf[B])(EqTerm);
EqTerm := Outermost(Linearity2Of[B])(EqTerm);
EqTerm := evala(Expand(EqTerm));
EqTerm := Outermost(V0IndependentOfY)(EqTerm);
EqTerm := Outermost(V1IndependentOfX)(EqTerm);
EqTerm := Outermost(LinearityOf[B])(EqTerm);

# Extra ad hoc factorization:
factor1 := [a*partial(u,x(i))*X_+a*partial(u,x(i))*Y_,a*partial(u,x(i))*(X+Y)];
EqTerm := Outermost(factor1)(EqTerm);

# Ad hoc inverse linearity rule.
factor2 := [B(X_)+B(Y_), B(X+Y)];
EqTerm := Outermost(factor2)(EqTerm);

# Step 3
EqTerm := Outermost(ApproximationB[1])(EqTerm);
EqTerm := ConvergenceStrategy(EqTerm);

# Step 4
EqTerm := Outermost(TwoScaleAdjoint)(EqTerm);

# Step 5
EqTerm := Outermost(ApproximationOfTHypo)(EqTerm);
EqTerm := Outermost(ApproximationT[3])(EqTerm);
EqTerm:=ConvergenceStrategy(EqTerm);

# Step 6
EqTerm := Outermost(LinearityOf[SUM])(EqTerm);
EqTerm := Outermost(GradientApprox)(EqTerm);
EqTerm:=ConvergenceStrategy(EqTerm);
quit;
```

E. Rule and transformation files

E.1. Domain integral and indefinite sum rules

```
# File:    integral.mpl.
# Content: Rules for integral and indefinite summation properties.

LinearityOf[Integral] := Linearity(2,x->y->x+y,Integral(Omega_,_,Z_));
Linearity2Of[Integral] := [Integral(Omega_,0,X_),0];

LinearityOf[partial] := Linearity(1,x->y->x+y,partial(_,B_));
Linearity2Of[partial] := Linearity2(1,partial(X_,Y_),Epsilon);

LinearityLambdaOf[Integral] := [
  X_*Integral(Omega_,Y_,Z_),
  Integral(Omega,X*Y,Z)];

LinearityOf[SUM] := Linearity(1,x->y->x+y,SUM(_,J_,K_));

SumZero := [SUM(0,X_,J_),0];

SumInteger := [
  SUM(A_*B_,J_,D_),
  A*(SUM(B,J,D)),
  DelayEval -> whattype(A)=integer];

# To group two sums into a single one:
SumFactor := [
  SUM(E_,I_,D_) + SUM(F_,I_,D_),
  DelayEval -> SUM(factor(E+F),I,D)];

# WARNING: Do not include the following rule in the same strategy as a
# factorization rule. May loop forever!
SumDistrib := [
  A_*(SUM(B_,J_,D_)),
  SUM(A*B,J,D)];

IntegralSumExchange := [
  Integral(M_,SUM(A_,J_,D_),B_),
  SUM(Integral(M,A,B),J,D)];

IntegrationStrategy :=
  STNormalizer(
    FailAsIdentity(
     LeftChoice([
       Outermost(LinearityOf[Integral]),
       Outermost(Linearity2Of[Integral]),
       Outermost(LinearityOf[SUM]),
       Outermost(SumZero),
       Outermost(SumInteger),
       Outermost([partial(0,X_),0])
     ])));
```

E.2. Green rules

```
# File:    Green.mpl.
# Content: Green rules.

GreenRule := [
  Integral(Omega_,V_*partial(U_,X_),Y_),
  Integral(partialBound([Omega,X]),U*V*Eta(X),s(X))
   -Integral(Omega,U* partial(V,X),Y)];

CondGreenRule := patt -> [
  Integral(Omega_,V_*partial(U_,X_),Y_),
  Integral(partialBound([Omega,X]),U*V*Eta(X),s(X))
   -Integral(Omega,U* partial(V,X),Y),
  DelayEval-> has(U,patt)];
```

E.3. Convergence rules

```
# File:    convergence.mpl.
# Content: Convergence theory and strategy.

OEpsilonSum := [
  BigO(I_,Epsilon) + BigO(J_,Epsilon),
  BigO(FreshIndex(),Epsilon)];

OEpsilonSumContext := [
  Y_+ BigO(I_,Epsilon) + BigO(J_,Epsilon),
  DelayEval -> BigO(FreshIndex(),Epsilon)+Y];

OEpsilonSUM := [
  SUM(BigO(I_,Epsilon),J_,D_),
  DelayEval -> BigO(FreshIndex(),Epsilon)];

OEpsilonSUMContext := [
  SUM(BigO(I_,Epsilon),J_,D_)+Z_,
  DelayEval -> BigO(FreshIndex(),Epsilon)+Z_];

OEpsilonIntegral := [
  Integral(Omega_, BigO(I_,Epsilon), X_),
  DelayEval -> BigO(FreshIndex(),Epsilon)];

# Controlled linearity
OEpsilonIntegralContext := [
  Integral(Omega_, BigO(I_,Epsilon)+E_, X_),
  DelayEval -> Integral(Omega, E, X)
    + BigO(FreshIndex(),Epsilon)];

# Controlled expansion
OEpsilonExpand := [
  A_ * (E_ + BigO(I_,Epsilon)),
  DelayEval -> A * E + A * BigO(I,Epsilon)];
```

```
# Only if A_ does not depend on Epsilon.
OEpsilonConst := [
  A_ * BigO(I_,Epsilon),
  DelayEval -> BigO(FreshIndex(),Epsilon)];

# Only if A_ does not depend on Epsilon.
EpsilonConst := [
  A_ * Epsilon,
  DelayEval -> BigO(FreshIndex(),Epsilon)];

ConvergenceStrategy :=
  STNormalizer(
    FailAsIdentity(
      LeftChoice([
        Outermost(OEpsilonSum),
        Outermost(OEpsilonSumContext),
        Outermost(OEpsilonSUM),
        Outermost(OEpsilonSUMContext),
        Outermost(OEpsilonExpand),
        Outermost(OEpsilonIntegral),
        Outermost(OEpsilonIntegralContext),
        Outermost(OEpsilonConst),
        OutermostNF(EpsilonConst)
      ])));
```

E.4. Two-scale method rules

```
LinearityOf[T] := Linearity(1,x->y->x+y,T(_));
LinearityBasic[TS] := Linearity(1,x->y->x+y,TS(_));

TwoScaleMult := [T(X_*Y_),T(X)*T(Y)];

TwoScaleAdjoint := [
  Integral(Omega_, TS(V_)*W_ , X_),
  Integral([TSTM(Omega), TSTm(Omega)], V*T(W), [X, y])];

TwoScaleAdjointInv := [
  Integral([TSTM(Omega_), TSTm(Omega_)], T(W_)*V_, [X_, Z_]),
  Integral(Omega, W*TS(V), X)];

PartialOfB := [
  partial(B(V_),x_(I_)),
  B(partial(V,x(I))) + 1 / Epsilon*B(partial(V,y(I)))];

% Array of two rules.
ApproximationTS := [
  [TS(V_),DelayEval -> B(V)+ BigO(FreshIndex(),Epsilon)],
  [TS(V_), DelayEval -> TS(V) + Epsilon * TS(y*partial(V,x))
              + Epsilon * BigO(FreshIndex(),Epsilon)]];
```

```
% Array of two rules.
ApproximationB := [
  [B(V_), DelayEval -> TS(V)+BigO(FreshIndex(),Epsilon)],
  x -> y -> j -> [
     B(V_),
     TS(V+Epsilon*SUM(y*partial(V,x),j,J))
      + Epsilon*BigO(FreshIndex(),Epsilon)
  ]
];

LinearityBasic[B] := LeftChoice([
  [B(0),0],
  Linearity(1,x->y->x+y,B(_))
]);

LinearityOf[B] :=
  STNormalizer(
    OutermostNF(
      IdentityAsFail(LinearityBasic[B])));

Linearity2Of[B] := [B(Epsilon*F_),Epsilon*B(F)];
LinearityOf[TS] := [A_*(1/Epsilon)*TS(V_),A*TS(1/Epsilon*V)];
```

E.5. Lemmas

```
# File:    lemmas.mpl.
# Content: Some useful lemmas.

# Zero- and first- order approximations of T.
ApproximationT := [
  [T(u), DelayEval -> u0 + BigO(FreshIndex(),Epsilon)],
  [T(u), u0 + Epsilon*u1 + Epsilon*SUM(y(j)*partial(u0,x(j)),j,J)
         + Epsilon*BigO('FreshIndex()',Epsilon)],
  [T(f), DelayEval -> f0 + BigO(FreshIndex(),Epsilon)]
];

IndependentOfX := [
  partial(y(I_)*A_,x(I_)),
  y(I)*partial(A,x(I))];

ApproximationOfTHypo := [
  T(a*partial(F_,X_)),
  DelayEval-> a0*T(partial(F,X)) + BigO(FreshIndex(),Epsilon)];

U0IndependentOfY := [
  partial(Const_*u0,y(i)),
  0];
```

E.6. Hypotheses

```
# File:    hypothesis.mpl.
# Content: Proof-dependent rules.
```

```
# Behavior of v on the boundary
OnBoundary[v] := [v(i),0];

BoundaryContext := B -> [
  Integral(Omega_,F_,X_),
  Integral(Omega,F,X),
  DelayEval-> has(Omega,B)];

# Integral of <w> is 0 on the boundary of <Domain>
IntegralOnBoundary:= w -> Domain ->
  Comp([
    BoundaryContext(Domain),
    OutermostNF([w,0]),
    OutermostNF([B(0),0])
  ]);

V0IndependentOfY := [
  partial(v0,y(i)),
  0
];

V1IndependentOfX := [
  partial(v1,x(i)),
  0
];

V0IndependentOfYIntegral := [
  Integral(Omega_,v0*E_,D_),
  v0*Integral(Omega,E,D)
];

PartialXjU0IndependentOfY := [
  Integral(Omega_,partial(u0,x(j))*E_,D_),
  partial(u0,x(j))*Integral(Omega,E,D)
];

PartialXkV0IndependentOfY := [
  Integral(Omega_,partial(v0,x(k))*E_,D_),
  partial(v0,x(k))*Integral(Omega,E,D)
];
```