Compositional Taylor Model Based Validated Integration

Kristjan Liiva
University of Edinburgh
K.Liiva@sms.ed.ac.uk

Paul B. Jackson
University of Edinburgh
Paul.Jackson@ed.ac.uk

Grant O. Passmore
Aesthetic Integration
grant.passmore@cl.cam.ac.uk

Christoph M. Wintersteiger
Microsoft Research Cambridge, UK
cwinter@microsoft.com

***Abstract*—We present a compositional validated integration method based on Taylor models. Our method combines solutions for lower dimensional subsystems into solutions for a higher dimensional composite system, rather than attempting to solve the higher dimensional system directly. We have implemented the method in an extension of the Flow* tool. Our preliminary results are promising, suggesting significant gains for some biological systems with nontrivial compositional structure.**

***Keywords*-validated integration, formal verification, Taylor models, nonlinear systems**

I. Introduction

Numerical solutions to Initial Value Problems (IVP) are of paramount importance for the analysis of hybrid and continuous systems. Typically, approximations like simulation are used to obtain these solutions. However, they provide only estimates that are 'close' to solutions, sometimes with unreliable, or even exponentially growing, error bounds. This means that they are not necessarily the best fit for use for the purposes of formal verification. Validated integration (VI) methods present an alternative that is more in tune with the needs of formal verification as they compute guaranteed bounds for the flow of ODEs, usually at the cost of increased computational complexity.

The field of VI was founded by Moore in the 60s [15] and early methods simply used interval arithmetic to address the roundoff errors and discretisation in computations. However, these methods frequently suffer from excessive imprecision due to coarse over-approximation. VI first encloses the flow of a set of ODEs in interval boxes in a fixed coordinate system. Often the dynamics of the system involve nonlinearity or rotation which greatly increases the size of the interval boxes. This is called the *wrapping effect*. Further, the relationships between variables may be lost when they are represented by intervals. This is known as the *dependency problem*.

An important next step in VI was the development of the parallelepiped method. The parallelepiped method attempts to minimise the wrapping effect by using a moving coordinate system [15]. Unfortunately, this approach faces challenges of stability with respect to long integration times, since the rotation is determined by a matrix that may become increasingly ill conditioned. Lohner's QR method [12] is an extension of the parallelepiped method that makes use of QR decomposition to find coordinate transformations that are stable. Despite these advances in tackling the wrapping effect, both the parallelepiped and QR methods are still prone to ineffeciencies rooted in the dependency problem.

To address these shortcomings, Berz and Makino introduced Taylor model (TM) based VI [2], [13]. In their approach, the flow of ODEs is handled by Taylor model arithmetic, which combines both symbolic computations and interval arithmetic. The core idea of the Taylor model based integration is that the majority of the flow of the ODEs is represented by a polynomial and all roundoff and discretisation errors are pushed into a remainder interval. The symbolic part is free of the dependency problem, and while the interval remainder part still suffers from it, its effect is reduced since the width of the interval box is smaller. A downside of the naive Taylor model approach is that the remainder interval can never decrease, and thus the impact of the dependency problem grows with integration time. To mitigate this, Berz and Makino developed preconditioned Taylor model based integration [14]. In preconditioned methods the solution is represented by a composition of TMs. The integration is only concerned with one of them and at each integration step, a model that minimises the wrapping effect and dependency problem is chosen.

Our present work began with the evaluation of various validated integration tools (CAPD [18], VSPODE [11], Flow* [4]) on a number of examples, many modelling chemical reactions in biological systems. While the Taylor-model based tools (VSPODE and Flow*) performed better, we encountered significant problems. In many cases the validated integration diverged well before reaching time limits of interest. When we altered integration parameters to increase accuracy and extend the integration time, integration became very slow, taking hours. We noted that the ODE systems arising from chemical reactions have considerable compositional structure. In this paper, we describe how we exploited this structure in order to improve the performance of Taylor-model-based validated integration that uses preconditioning.

The structure of the paper is as follows. In Section II, we present preliminaries. In Section III, we will give a description of validated integration methods. In Section IV, we describe the properties of systems suitable for the compositional approach, adapt validated integration methods to the compositional setting and examine the interesting case

of the preconditioned compositional approach in more detail. In Section V, we provide our experimental results together with analysis and we conclude with observations and some open problems in Section VI.

A. Related Work

Chen and Sankaranarayanan present an approach that is close to the present work [5]. They take a similar approach of partitioning the system into smaller systems and then solving them separately. The key difference with our work is that they are partitioning the system into hybrid systems based on the range of effects that the subsystems have on each other. In their setting, the effect of one sub-system on another is abstracted to an interval, while we track the effect symbolically with Taylor models.

Other hybrid and continuous system analysis tools include dReach [9], CAPD [18], iSAT [6], KeYMaera [17], VNODE-LP [16] and VSPODE [11]. These tools also utilise interval arithmetic (and some include forms of modular reasoning), but all utilise techniques quite different from our work.

A connection with decomposing a system can also be made with hybridisation tools such as Nltoolbox [19] and SpaceEx [7]. Hybridisation [1] is essentially splitting up a continuous state space into regions s.t. each region becomes a state in a hybrid system, with the dynamics of each region approximated linearly.

II. Preliminaries

A. Interval Arithmetic

Many rigorous computations involving reals cannot be carried out directly using floating point numbers. However, real numbers can be enclosed in *intervals* with floating point endpoints, and rigorous arithmetical operations may be implemented for computing with these intervals. A real number $x \in \mathbb{R}$ may be represented by a floating point interval $[\underline{x}, \overline{x}]$ s.t. $\underline{x} \leq x \leq \overline{x}$. We denote the set of all real intervals as $\mathbb{IR}$. Interval arithmetic operations must satisfy the obvious *enclosure* property

$$[\underline{x}, \overline{x}] \; op \; [\underline{y}, \overline{y}] \supseteq \{x \; op \; y \mid \underline{x} \leq x \leq \overline{x} \wedge \underline{y} \leq y \leq \overline{y}\}$$

where, e.g., $op \in \{+, -, \times, \div\}$. An interval lifting of a real operation $\circ\mathrm{p}$ satisfying this enclosure property is known as an *interval extension* [8].

Unnecessary over-approximation is often introduced when evaluating interval extensions. For example, consider the naive interval extension of the function $f(x) = x - x$ evaluated over the interval $[-1, 1]$. With the standard interval extension for subtraction, this will be computed as $[-1, 1] - [-1, 1] = [-2, 2]$, yet it is obvious that $f([-1, 1])$ is actually $[0, 0]$. This imprecision is a manifestation of the dependency problem.

B. Taylor Model Arithmetic

A k-th order n-parameter Taylor model (TM) over a domain $D \subset \mathbb{R}^n$ is a pair $(p, \mathbf{i})$ of a k-th order n-variable polynomial p and a *remainder interval* $\mathbf{i} \in \mathbb{IR}$. It approximates an $\mathbb{R}$-valued function $f : D \subset \mathbb{R}^n \to \mathbb{R}$ when $\forall x \in D. f(x) \in p(x) + \mathbf{i}$. We refer to the variables of a TM as *parameters* in order to avoid confusion with the variables of the ODE systems being solved.

We define arithmetic operations on TMs as described by Makino [13]. For addition and subtraction, one may operate on the polynomial and remainder parts separately. For k-th order TMs $(p_1, \mathbf{i}_1)$ and $(p_2, \mathbf{i}_2)$ both over domain D, addition and subtraction are defined as

$$(p_1, \mathbf{i}_1) \pm (p_2, \mathbf{i}_2) = (p_1 \pm p_2, \mathbf{i}_1 \pm \mathbf{i}_2) \; ,$$

resulting in k-th order TMs over domain D. Multiplication is slightly more complicated; it is defined as

$$(p_1, \mathbf{i}_1) \times (p_2, \mathbf{i}_2) = p_1 \times p_2 + p_1 \times \mathbf{i}_1 + p_2 \times \mathbf{i}_2 + \mathbf{i}_1 \times \mathbf{i_2} \; .$$

Since $p_1 \times p_2$ is of order $2k$, it is split into two parts $p_1 \times p_2 = p + p_e$, where p contains the terms of order up to k and p_e contains all other terms. Using the interval extension B of the polynomial part p we evaluate the polynomial over an interval and express multiplication as

$$\begin{aligned}(p_1, \mathbf{i}_1) \times (p_2, \mathbf{i}_2) = (p, B(p_e) + B(p_1) \times \mathbf{i}_1 + \\ B(p_2) \times \mathbf{i}_2 + \mathbf{i}_1 \times \mathbf{i_2}) \; .\end{aligned}$$

We commonly work with vectors of Taylor models $\vec{TM}$ and write typing statements such as $\vec{TM} : D \subset \mathbb{R}^n \to \mathbb{IR}^m$. We also call these vectors *Taylor models*. Sometimes we consider TMs extended to apply to interval-valued arguments, in which case we might write $\vec{TM} : D \subset \mathbb{IR}^n \to \mathbb{IR}^m$.

Composition $\vec{\mathcal{U}} \circ \vec{\mathcal{V}}$ of two Taylor models $\vec{\mathcal{U}} : D_1 \subset \mathbb{R}^q \to \mathbb{IR}^r$ and $\vec{\mathcal{V}} : D_2 \subset \mathbb{R}^p \to \mathbb{IR}^q$ is defined by substituting the variables of $\vec{\mathcal{U}}$ with the corresponding elements of $\vec{\mathcal{V}}$ and using Taylor model arithmetic to put the result in the normal form $(\vec{p}, \mathbf{i})$. Composition is only valid if the image under $\vec{\mathcal{V}}$ of $\vec{\mathcal{V}}$'s domain is a subset of the domain of $\vec{\mathcal{U}}$, i.e. $\vec{\mathcal{V}}(D_2) \subseteq D_1$.

III. Validated Integration

Definition 1 (Extended Initial Value Problem): An instance of this problem has the form of an Initial Value Problem

$$\frac{d\vec{x}}{dt} = \vec{f}(\vec{x}), \quad \vec{x}_{init} = \vec{x}(t_{init}) \in \vec{X}_{init}, \quad t \in [t_{init}, t_{final}] \tag{1}$$

where the vector field $\vec{f} : \mathbb{R}^n \to \mathbb{R}^n$ is a sufficiently-smooth function, $\vec{X}_{init} \subseteq \mathbb{R}^n$ is a set of initial conditions and solutions are sought for $\vec{x}$ in the time interval $[t_{init}, t_{final}]$.

For any point $\vec{x}_{init} \in \vec{X}_{init}$ we have a classical IVP problem. We assume for each $\vec{x}_{init} \in \vec{X}_{init}$ and all time $t \in [t_{init}, t_{final}]$ that a solution for $\vec{x}$ exists and is unique.

For example, if $\vec{f}$ is Lipschitz continuous in neighbourhoods of all $\vec{x}_{init} \in \vec{X}_{init}$, then the Picard Lindelöf theorem ensures the existence of solutions in some closed time interval around t_{init}, and we then assume that $[t_{init}, t_{final}]$ is a subset of this interval.

Our goal is to find a set that encloses the IVP solutions. We call this enclosure set the *flow* or *flowpipe* of the EIVP.

We divide the time interval $[t_{init}, t_{final}]$ into a number of smaller intervals and describe the advance in time from the start to the end of one of these intervals as a *timestep*. In the following we focus on how, given an enclosure of the solution values at the start of a timestep, one can find a flowpipe over the step interval (sometimes called a *flowpipe segment*) and hence an enclosure of the solution values at the end of the step. This end enclosure then serves as an enclosure for the start of the next timestep.

The scheme used in Flow* for computation of a flowpipe segment is based on versions of the Picard operator. These operator versions are used to compute successively better approximations to a solution or sets of solutions over a timestep. For convenience let us now use a time variable t to measure time since the start of a timestep and let δ be the timestep size. We then have $0 \leq t \leq \delta$. Let $\Delta = [0, \delta]$.

The basic Picard operator takes the form

$$\mathbb{P}_{\vec{f}}(\vec{g})(t) = \vec{x}_0 + \int_0^t \vec{f}(\vec{g}(s))ds$$

where $\vec{g}(t)$ is some approximation to a solution that takes on value $\vec{x}_0$ at the timestep start when $t = 0$. $\mathbb{P}_{\vec{f}}(\vec{g})$ is a better approximation of the solution. Assuming we have the conditions guaranteeing the existence and uniqueness of a solution, one can show using the Banach fixed-point theorem that this solution is the unique fixed-point of the Picard operator $\mathbb{P}_{\vec{f}}()$, and that the approximations obtained by iterating application of the Picard operator converge to this solution.

For validated integration we use an interval generalisation $\mathbb{P}^I_{\vec{f}}()$ of this basic operator that for example allows $\vec{x}_0$, $\vec{f}$ and $\vec{g}$ to take on interval values in $\mathbb{IR}^n$ and allows $\vec{f}$ and $\vec{g}$ to take interval arguments in $\mathbb{IR}^n$ and $\mathbb{IR}^n \times \mathbb{IR}$ respectively ($\vec{g}$ takes an extra time argument). The initial condition $\vec{x}_0$ is modelled using a TM with domain $D \subset \mathbb{R}^n$ and functions $\vec{g}$ and $\mathbb{P}_{\vec{f}}$ using TMs with domain $D \times \Delta$. One choice for D is the set of initial conditions $\vec{X}_{init}$ at the beginning of the validated integration. Later in this section, when we discuss the concept of preconditioning, we remark on other choices for D. We also express $\vec{f}$ using a TM which encloses the actual $\vec{f} : \mathbb{R}^n \to \mathbb{R}^n$. Making the dependence on parameters $\vec{a} \in D$ explicit, the interval Picard operator takes the form:

$$\mathbb{P}^I_{\vec{f}}(\vec{g})(\vec{a}, t) = \vec{x}_0(\vec{a}) + \int_0^t \vec{f}(\vec{g}(\vec{a}, s))ds.$$

A TM for the flowpipe segment over a timestep is computed in three phases.

1) Iterate the Picard operator k times starting with an initial $\vec{g}(\vec{a}, t) = \vec{x}_0(\vec{a})$ in order to compute a k^{th}-order TM with the correct polynomial part $\vec{p}$. Further iterations do not change this polynomial part. For efficiency these computations ignore tracking the interval parts of TMs and discard monomials of order greater than k.
2) Guess an interval $\mathbf{i}$ with the hope that the TM $(\vec{p}, \mathbf{i})$ encloses all the solutions to the EIVP that have start values at time 0 in the interval defined by the TM $\vec{x}_0$. Compute $(\vec{p'}, \mathbf{i'}) = \mathbb{P}^I_{\vec{f}}((\vec{p}, \mathbf{i}))$ and derive interval enclosures of all monomials in $\vec{p'}$ of order greater than k in order to arrive at a k^{th} order TM $(\vec{p}, \mathbf{i''})$ that encloses $(\vec{p'}, \mathbf{i'})$. Using the Banach fixed point theorem it can be shown that if $\mathbf{i''} \subseteq \mathbf{i}$, then the TMs $(\vec{p}, \mathbf{i})$ and $(\vec{p}, \mathbf{i''})$ both enclose all the EIVP solutions.
   If the inclusion test $\mathbf{i''} \subseteq \mathbf{i}$ fails, an alternate $\mathbf{i}$ is searched for by successively trying larger guesses.
3) Tighter enclosures of the set of solutions can be found by iterating the Picard operator on $(\vec{p}, \mathbf{i''})$. As observed in the PhD thesis of the Flow* implementer [3], efficiency gains can be made in the application of the Picard operator here, since the polynomial arithmetic computations are largely the same as in the previous phase and so do not have to be repeated.
   In Flow*, these iterations are terminated when the decrease in the remainder interval $\mathbf{i}$ size on an iteration is less than some threshold.

After the last phase, the resulting TM is evaluated at the timestep end time δ in order to give a TM that encloses the solution values at the timestep end and that can be used as an enclosure of the solution values at the start of the next timestep.

Preconditioning [14] is an important technique for reducing the effects of the wrapping problem in TM based integration and controlling the growth of remainder intervals. Its use can also lead to significant improvements in performance: indeed in our experience this benefit seems in most cases more important than reductions in the effects of the dependency problem.

The basic idea is to express the TM for a flowpipe segment as a composition $\vec{\mathcal{U}} \circ \vec{\mathcal{V}}$ of a left model $\vec{\mathcal{U}}$ and a right model $\vec{\mathcal{V}}$. The right model $\vec{\mathcal{V}}$ acts as a transformation of the domain D_r over which the parameters of the right model take values. In our work the domain D_r is always an interval box for the set of initial conditions $\vec{X}_{init}$. During the Picard iterations for a validated integration step, the Picard operator does not operate on the composition $\vec{\mathcal{U}} \circ \vec{\mathcal{V}}$. Rather, it just operates on the left $\vec{\mathcal{U}}$ and the right model $\vec{\mathcal{V}}$ is held constant. The left TM then has parameters that take values from the transformed domain $\vec{\mathcal{V}}(D)$, as well as a time parameter with values from $[0, \delta]$.

One particularly-simple right model is an affine trans-

formation that maps $\vec{X}_{init}$ to the interval box $[-1,1]^n$. Having intervals for each coordinate centred around zero ensures that interval arithmetic involved in calculating interval bounds on higher order monomials produces better results than when coordinate intervals are not centred around zero.

In general the left and right models are both adjusted, keeping their composition an overapproximation, between integration steps. With *identity preconditioning*[1], the new left model is set to the identity TM and the new right model is set to the composition of the old left and right models. This yields significant performance gains as the Picard operator works on much smaller polynomials than in the non-preconditioned case. As the left model at the start of each integration step is the identity, the domain D_l of the left model is the set $\vec{X}_0$, an enclosure of the IVP solution values at the step start.

In general the preconditioning transformation has form[2]

$$\begin{aligned} \vec{\mathcal{U}} \circ \vec{\mathcal{V}} &= [\lambda \vec{a}.\ \vec{c} + \vec{C}^* \vec{a} + \vec{N}^*(\vec{a})] \circ \vec{\mathcal{V}} \\ &= (\lambda \vec{a}.\ \vec{c} + \vec{C}\vec{a} + [0,0]^n) \circ \\ &\quad \left\{ \vec{C}^{-1}(\lambda \vec{a}.\ \vec{C}^* \vec{a} + \vec{N}^*(\vec{a})) \circ \vec{\mathcal{V}} \right\} \\ &= \vec{\mathcal{U}}' \circ \vec{\mathcal{V}}' \end{aligned} \tag{2}$$

where the old left TM $\vec{\mathcal{U}} = \lambda \vec{a}.\ c + C^* \vec{a} + N^*(\vec{a})$ consists of constant part c, linear part $C^* \vec{a}$ and non-linear and remainder interval part $N^*(\vec{a})$, and C is the desired linear part for the new left TM $\vec{\mathcal{U}}'$. Here we use lambda abstraction to indicate how Taylor models act as operators. Usually, the variables bound by such a lambda abstraction correspond to the Taylor model parameters. Typically C is a function of C^*. With identity preconditioning, C is the identity matrix. With *parallelepiped preconditioning*, C is C^*. With *QR preconditioning*, C is the matrix Q of the QR factorization of the matrix obtained by sorting the columns of C^* by size in descending order. Makino and Berz [14] discuss the relative merits of these different preconditioning schemes.

IV. Compositional Validated Integration

Let us define a *dependency graph* $G = (\vec{x}, \rightarrow)$ with the variables $\vec{x}$ of the ODE system (1) as vertices and and an edge $y \rightarrow x$ whenever the ODE vector field $\vec{f}$ makes $\frac{dx}{dt}$ directly depend on y. If the specific ODE for $\frac{dx}{dt}$ is of form $\frac{dx}{dt} = f_x(y_1, \ldots, y_m)$ (with $\{y_1, \ldots, y_m\} \subseteq \vec{x}$), then $\rightarrow$ includes an edge $y_i \rightarrow x$ for each y_i.

This dependency graph tells us how $\frac{dx}{dt}$ at some time t might be influenced by the values of system variable $z \in \vec{x}$ at times $t' \leq t$. Specifically, $\frac{dx}{dt}$ can be influenced by z just when $z \rightarrow^* x$, where $\rightarrow^*$ is the reflexive transitive closure of $\rightarrow$. We sometimes say that z is an *influencer* of x when $z \rightarrow^* x$. If $z \rightarrow^* x$ does not hold for some $x, z \in \vec{x}$, then the time evolution of x is independent of z and we have the option of finding a solution for x for all time before we start on finding a solution to z. On the other hand if we have both $z \rightarrow^* x$ and $x \rightarrow^* z$, then the system variables x and z are mutually dependent and we have to advance their solutions together.

In general, the solutions of the system variables in each strongly-connected component of G have to be advanced together. Every variable is a member of exactly one strongly-connected component and the components form a partition of the set of all system variables. Consider a directed graph on these components with an edge from a component A to component B when there is an edge from some variable in A to some variable in B. This component graph is acyclic. Any topological sort of this component graph defines a suitable ordering in which we can completely solve each component in turn. This observation motivates our compositional approach.

As explained in the previous section, the core computation is iteration of the Picard operator. Let us focus on the effect of the iteration on a particular system variable $x \in \vec{x}$ with $\frac{dx}{dt} = f_x(y_1, \ldots, y_m)$ as above. The successive approximations $x^{(j)}$ of the solution for x computed by the Picard operator are:

$$x^{(j)}(\vec{a}, t) = x_0(\vec{a}) + \int_0^t f_x(y_1^{(j-1)}(\vec{a}, s), \ldots, y_m^{(j-1)}(\vec{a}, s))ds$$

with $x^{(0)}(\vec{a}, t) = x_0(\vec{a})$. Here for simplicity we blur the distinction between the different phases discussed previously.

Now, let us assume that each system variable x depends on a vector of parameters $\vec{a}_x$ that is a subvector of the vector of all parameters $\vec{a}$, where if $z \rightarrow^* x$, then $\vec{a}_z \subseteq \vec{a}_x$ ($\vec{a}_z$ is a subvector of $\vec{a}_x$). We can then write the successive approximations of the solution for x as $x^{(j)}(\vec{a}_x, t)$ rather than $x^{(j)}(\vec{a}, t)$. Specifically, we can write:

$$x^{(j)}(\vec{a}_x, t) = x_0(\vec{a}_x) + \int_0^t f_x(y_1^{(j-1)}(\vec{a}_{y_1}, s), \ldots, y_m^{(j-1)}(\vec{a}_{y_m}, s))ds$$

with $x^{(0)}(\vec{a}_x, t) = x_0(\vec{a}_x)$. Note here that all the $y_i^{(j-1)}$ take as arguments $\vec{a}_{y_i}$ which are subvectors of $\vec{a}_x$, so the right-hand side of these equations only depend on parameters in $\vec{a}_x$ and we are justified in writing $x^{(j)}(\vec{a}_x, t)$ on the left. These parameters $\vec{a}$ might be used to specify the value of the system variables $\vec{x}$ at the initial integration time t_{init}, with one parameter $a \in \vec{a}$ for each $x \in \vec{x}$. However this is not necessary. For example, we need not introduce a parameter for a system variable when its initial value is fixed and not varying over some interval.

One major performance gain of the compositional approach is a consequence of the TMs for each system variable solution having fewer parameters. Following the Flow* implementation, we use a dense representation for

[1] When ignoring constant part of both left and right model.

[2] Due to simplicity we leave out the details of the scaling used to guarantee that the range of right model is subset of unit interval box.

monomials: polynomial computations for a given component are carried out with all monomials over $\vec{a}_x$ where each monomial is represented using a vector of parameter exponents. This is possible because the TMs for all x in a component have the same parameters. In order to use TMs from one component A in another component A', we have to convert the monomials from the representation appropriate for A to that appropriate for A'. This conversion cost is an overhead of using a dense representation. The cost could be avoided by switching to a sparse representation (monomials as sequences of pairs of parameter names and positive exponents), but we estimate the performance benefit from a dense representation well outweighs this cost. The performance gain is because the monomial operations such as copying and multiplying have run-time proportional to the size of the monomials.

In the worst case, the dependency graph has a single strongly-connected component and no gain is possible. However, with many ODE systems, for example those derived from chemical reaction networks, many components have significantly fewer parameters than the total number, so significant peformance gains are possible.

One way to sequence the computations of the $x^{(j)}$ is to mimic what happens in the non-compositional case and compute the $x^{(j)}$ for all $x \in \vec{x}$ for a given j before moving on to the $x^{(j+1)}$. Instead in our implementation we choose to complete the computation of the best approximation of the each variable in some component before moving on to any following components in the component dependency graph. This component by component approach gives us the freedom to iterate different numbers of times for each component. When computing $x^{(j)}$, we use $y_i^{(j-1)}$ for those y_i in the same component as x, but use the final approximations $y_i^{(\mathrm{fin}_{y_i})}$ for those y_i in ancestor components. This approach is sound because the final approximations are always tighter enclosures of the solutions than the $y_i^{(j-1)}$.

How are dependencies affected by preconditioning? Let's assume that one of approximations $\vec{x}^{(j)}$ to the ODE solution is represented by the preconditioned pair of TMs $\vec{\mathcal{U}} \circ \vec{\mathcal{V}}$. As noted earlier, with preconditioning the successive iterations that compute better approximations of the solution for a flowpipe step only operate on the left TM $\vec{\mathcal{U}}$ and during these iterations the right TM $\vec{\mathcal{V}}$ stays fixed. Because of composition, the elements of $\vec{\mathcal{U}}$ only depend on some parameters; specifically element $\mathcal{U}_x$ depends only on $\vec{a}_x$.

Let us now consider how the preconditioning step itself is affected by composition. Let $\vec{\mathcal{U}} \circ \vec{\mathcal{V}}$ be the final Taylor model produced by an integration step. The Taylor model for a single variable x is given by $\mathcal{U}_x \circ \vec{\mathcal{V}}$ where $\mathcal{U}_x$ is the element of $\vec{\mathcal{U}}$ corresponding to x. Narrowing equation (2) to $\mathcal{U}_x \circ \vec{\mathcal{V}}$, we have

$$\mathcal{U}_x \circ \vec{\mathcal{V}} = \overbrace{(\lambda \vec{a}.\ c_x + C_x \vec{a} + [0,0])}^{\mathcal{U}'_x} \circ \overbrace{(\vec{C}^{-1}(\lambda \vec{a}.\ \vec{T}^*(\vec{a})) \circ \vec{\mathcal{V}})}^{\vec{\mathcal{V}}'}$$

where C_x is the row of the matrix $\vec{C}$ corresponding to x and $\vec{T}^*(\vec{a}) = \vec{C}^*\vec{a} + \vec{N}^*(\vec{a})$ is the sum of the linear and nonlinear part.

The new left TM element $\mathcal{U}'_x = \lambda \vec{a}.\ c_x + C_x\vec{a} + [0,0]$ acts as the initial condition for the next integration step. We need to ensure that $\mathcal{U}'_x$ depends only on $\vec{a}_x$. C_x has the form

$$\begin{array}{cc} & \begin{array}{cccc} a_1 & a_2 & \dots & a_n \end{array} \\ x & [C_{x,a_1} \quad C_{x,a_2} \quad \cdots \quad C_{x,a_n}] \end{array}$$

so we need to have the C_{x,a_i} non-zero only when $a_i \in \vec{a}_x$. We call preconditioning methods using such $\vec{C}$ *left model compositional* methods. Parallelepiped preconditioning is an example of a left model compositional method.

From now on we are only concerned with left model compositional methods and we explore when compositionality can also allow us to reduce the parameters required by right TM elements. Consider the right TM

$$\vec{\mathcal{V}}' = \vec{C}^{-1}(\lambda \vec{a}.\ \vec{T}^*(\vec{a})) \circ \vec{\mathcal{V}}$$

and let us focus on only the element corresponding to x

$$\mathcal{V}'_x = C_x^{-1}(\lambda \vec{a}.\ \vec{T}^*(\vec{a})) \circ \vec{\mathcal{V}}.$$

If C_x^{-1} has the form

$$\begin{array}{cc} & \begin{array}{cccc} a_1 & a_2 & \dots & a_n \end{array} \\ x & [k_{x,a_1} \quad k_{x,a_2} \quad \cdots \quad k_{x,a_n}] \end{array}$$

we can write

$$\mathcal{V}'_x = (\lambda \vec{a}.\ k_{x,a_1} T^*_{x_1}(\vec{a}) + \cdots + k_{x,a_n} T^*_{x_n}(\vec{a})) \circ \vec{\mathcal{V}}.$$

Since we are now assuming the preconditioning method is left model compositional, we know that $T^*_{x_i}$ actually only depends on parameters $\vec{a}_{x_i}$. We make this explicit by introducing a version $\tilde{T}^*_{x_i}$ of the operator $T^*_{x_i}$ that takes as argument $\vec{a}_{x_i}$ rather than $\vec{a}$, and write

$$\mathcal{V}'_x = (\lambda \vec{a}.\ k_{x,a_1} \tilde{T}^*_{x_1}(\vec{a}_{x_1}) + \cdots + k_{x,a_n} \tilde{T}^*_{x_n}(\vec{a}_{x_n})) \circ \vec{\mathcal{V}}$$

We call left model compositional preconditioning methods *fully compositional* preconditioning methods if the k_{x,a_i} are non-zero only if x_i is an influencer of x. Identity preconditioning is an example of a fully compositional preconditioning method.

Assuming that the preconditioning method is fully compositional, we have

$$\mathcal{V}'_x = (\lambda \vec{a}.\ k_{x,a_{z_1}} \tilde{T}^*_{z_1}(\vec{a}_{z_1}) + \cdots + k_{x,a_{z_p}} \tilde{T}^*_{z_p}(\vec{a}_{z_p})) \circ \vec{\mathcal{V}}$$

where the z_i are all the influencers of x ($z_i \to^* x$) and a_{z_i} is the parameter associated with z_i. Keeping in mind that $\vec{a}_{z_i} \subseteq \vec{a}_x$, we can create lifted versions $\tilde{T}^{*x}_{z_i}$ of the operators $\tilde{T}^*_{z_i}$ that take arguments $\vec{a}_x$ rather than $\vec{a}_{z_i}$. We then have

$$\mathcal{V}'_x = (\lambda \vec{a}_x.\ k_{x,a_{z_1}} \tilde{T}^{*x}_{z_1}(\vec{a}_x) + \cdots + k_{x,a_{z_p}} \tilde{T}^{*x}_{z_p}(\vec{a}_x)) \circ \vec{\mathcal{V}}\Big|_x \quad (3)$$

where $\vec{\mathcal{V}}\Big|_x$ is $\vec{\mathcal{V}}$ restricted to exactly the $\mathcal{V}_z$ such $z \rightarrow^* x$.

We are now ready to show how a particular compositional structure of the right TM is preserved by preconditioning. Let us assume each right TM element $\mathcal{V}_x$ has a vector of parameters $\vec{b}_x$ where if $z \rightarrow^* x$ then $\vec{b}_z \subseteq \vec{b}_x$. From equation (3) above, the $\vec{b}$ parameters involved in $\mathcal{V}'_x$ are

$$\bigcup_{z \rightarrow^* x} \vec{b}_z.$$

But since we have that all such $\vec{b}_z \subseteq \vec{b}_x$, the parameters involved in $\mathcal{V}'_x$ are just $\vec{b}_x$.

If a preconditioning method is neither left model compositional or fully compositional then we call it a *non-compositional* preconditioning method. QR preconditioning is an example of non-compositional method.

V. Experiments

To assess the performance of our approach, we implemented a prototype by extending the (non-compositional) Flow* [4] version 2.0.0 with our compositional solving method. Our prototype uses the same parameters and data-structures as Flow*, with the exception of the remainder estimation, which we compute dynamically instead of statically. All experiments were performed on a workstation running Scientific Linux 7.4 with Intel 3.10GHz i5-2400 and 8GB memory.

Our prototype computes the flows of all ODEs for the specified amount of time or until a suitable remainder cannot be found in the validation step.

We present two classes of experiments. In the first class, we compare performance of preconditioning methods by examining (a) how far they manage to integrate the system (or if they managed to complete the integration time goal at all), and (b) then comparing the widths of the flows. In the second class, we use the compositional approach in solving the system with the aforementioned determined "best" preconditioning method. As described in detail below, if a system is non-compositional by nature, we make it compositional "artificially" by considering a composite system composed of copies of the non-compositional system.

A. Preconditioning method experiments

We use our tool to compute flows for 5 different systems. The systems are Lotka-Volterra (Lotka-Volterra equations), AND-Gate (computes the logical AND-function), Sq-Deg (nonlinear artificial system with no dependencies), Lin-Dep (linear artificial system with cascading dependencies), AND-OR (OR-function of two AND-gates).

Our goal is to compute flows until a specified integration time is reached. Sometimes that is not possible because the witdh of the enclosure expands exponentially. If that is the case then our tool halts, reporting that it can only compute the flow up to the given time.

The first type of experiments we did were concerned with determining the most suitable preconditioning method for a given system. Given two preconditioning methods A and B, we say that A is *better* than B if (i) A integrates further than B, or (ii) A and B integrate for the same amount of time but A produces a flow that can be bounded by a smaller interval box.

We present the data for preconditioning methods in Table I. Based on this data, we decided to use parallelepiped preconditioning with Lotka-Volterra and identity preconditioning with all other systems[3].

B. Compositional vs non-compositional

This section is concerned with comparing the compositional and non-compositional approaches.

We note that two of our systems (Lotka-Volterra and Squared-Degration) are not compositional in nature and we simulate compositionality by using identical copies of the system in parallel with no dependencies (10 copies for Lotka-Volterra and 20 copies for Squared-Degration).

For each example system, we pick the best preconditioning method as determined by the previous section. When we pick identity preconditioning we present data for compositional integration and for compositional integration together with compositional preconditioning. With parallelepiped preconditioning we only use compositional integration.

Since our compositional flow is practically identical to the non-compositional one, we refrain from presenting the flow itself, but report instead the time needed to compute it and the time spent in key parts of the algorithm. These times correspond to

- total time needed to solve the system (total),
- finding the polynomial in the Picard iteration (polynomial),
- validating the existence and uniqueness of the solution (validating),
- tightening the remainder interval (refinement),
- mapping before preconditioning (mapping 1),
- preconditioning the left and right model (preconditioning),
- mapping after preconditioning (mapping 2).

Table II presents a summary of all experimental results.

Our current setup treats the non-compositional approach as having everything in a single component. This has the effect of adding some insignificant overhead in the non-compositional approach (mapping 1 and mapping 2 for non-compositional rows in Table II).

Another general observation to be made is that in fully compositional methods we need a mapping from dependencies (mapping 1). In partially compositional and non

[3]Note that with our current implementation the parallelepiped preconditioning wasn't applicable to the system called AND-Gate

Example	Dim	Order	Time	Step	Preconditioning method					
					QR		Pa		Id	
					IP	Wid	IP	Wid	IP	Wid
Lotka-Volterra	2	4	10	0.2	10	0.851	10	**0.769**	10	3.98
AND-Gate	7	5	1000	10	1000	0.00781	-	-	1000	**0.00771**
Sq-Deg[3]	1	5	1	0.1	100	**0.0109**	100	**0.0109**	100	**0.0109**
Lin-Dep[3]	2	5	1	0.1	1	**0.466**	1	**0.466**	1	**0.466**
AND-OR	30	2	40	0.05	40	0.0401295	40	0.0250833	40	**0.0229715**

Table I
PRECONDITIONING EXPERIMENTS

Abbreviations: Dim : Dimension of the system, Step : Time step of the integration, QR : QR preconditioning, ID : identity preconditioning, Pa : parallelepiped preconditioning, IP : integration progress, Wid: width of the flow for first variable

Example	Dim	#C	Pre Met	Comp	Times						
					Total	Integration			Preconditioning		
						Poly	Val	Ref	Map 1	P W/O M	Map 2
Lotka	20	10	PA	LC	**59.03**	**8.26**	**18.18**	**7.1**	0.91	**22.04**	1.52
				NC	127.58	10.5	45.2	45.78	**0.76**	22.29	1.94
Sq-Deg	20	20	ID	FC	**6.54**	1.36	2.73	**1.49**	**0**	**0.83**	0
				LC	7.32	**1.35**	**2.57**	1.51	0.19	1.3	0.31
				NC	22.06	1.9	7.99	10.14	0.14	1.4	0.32
Lin-Dep	20	20	ID	FC	0.61	0.14	**0.13**	**0.09**	0.12	**0.06**	0
				LC	**0.58**	0.12	0.15	0.13	0.02	0.08	0.04
				NC	1.25	**0.1**	0.42	0.59	**0.01**	0.1	0.01
AND	7	3	ID	FC	36.4	12.3	14.54	1.79	0.14	7.31	0
				LC	36.19	11.92	14.43	**1.76**	0.16	7.28	0.13
				NC	**31.53**	**6.58**	**14.35**	3.1	**0.09**	**7.17**	0.06
AND-OR	30	22	ID	FC	**61.76**	**3.99**	**28.17**	**15.09**	2.7	**8.36**	0
				LC	69.99	4.19	28.98	15.25	**0.81**	11.92	6.44
				NC	348.96	5.24	124.61	199.12	0.88	11.53	6.57

Table II
COMPOSITION EXPERIMENTS

Abbreviations: Dim : dimension, #C : number of components, Pre Met : preconditioning method, Comp : composition type, FC : fully compositional, LC : left model compositional, NC : non-compositional, Map 1 : mapping 1, Precond : preconditioning without mapping, Map 2 : mapping 2, Poly : polynomial, Val : validating, Ref : refinement.

compositional cases we need to map components before and after the preconditioning phase (mapping 1 and mapping 2).

In general we make the following observations:

- Compared to the non-compositional approach, fully compositional and left model compositional methods perform similarly better on the integration phase.
- Compared to the fully compositional approach, left model compositional and non-compositional methods perform similarly worse on the preconditioning phase.

The massive speedup for the compositional approaches to the Lotka-Volterra model reflect the fact that the benchmark was in a sense crafted to be suitable for composition: Each part of the integration performs better with composition. Although we do not gain much in the preconditioning phase, the total computation time is dominated by integration, so we still obtain a significant performance gain overall.

The Squared-degradation system is another system somewhat artificially engineered to be suitable for composition. Since there are no dependencies between components, we do not need to do any mapping with compositional preconditioning and we achieve significant preconditioning gains in the compositional setting.

The linearly dependent system is compositional but in some ways unsuitable for our fully compositional approach. At a high level, this is because of a specific component with the property that all components that are not in its dependencies depend on it. As a result there is a lot of time spent on mappings in the preconditioning phase.

Turning to the destiffened AND-Gate, we observe that the time spent computing the polynomial during Picard iteration is significantly larger with composition. We conjecture this is the case because, in our implementation, each Picard operator iteration for the system variables in some component makes use of the final most-complex form of the polynomials for other system variables that these ones immediately depend on. In contrast, in the non-compositional Picard iterations, the polynomials of all system variables are refined in lock-step together, so the polynomial computations are simpler. This design choice in our implementation is not essential; it is possible in a compositional scheme for the

iterations for every system variable to advance together. We will investigate this in future work.

Finally, we can see that composition yields significant performance gains on the AND-OR-Gate model, with the fully compositional approach taking less than 20% of the time of the non-compositional method. This can be explained by the fact that most of the system can be partitioned into two independent parts which are particularly suitable for composition[4] on the integration phase. These independent system share a key characteristic with the linearly dependent system in that there are a high number of components with many dependencies, which results in a performance hit for compositional preconditioning.

VI. Conclusion

We have presented a compositional approach for Taylor model based validated integration. Through an implementation in the Flow* tool, we have observed that our compositional approach yields significant performance gains in the analysis of some compositional biological systems. These improvements are mostly in the 'integration' phase, but we also observe improvements during the preconditioning phase in some cases of identity preconditioning. We find these results promising and plan to continue this work in the following two directions: (i) adapting compositional versions of more preconditioning methods, and (ii) relaxing some restrictions so that the method is applicable to a wider class of continuous systems.

References

[1] Eugene Asarin, Thao Dang, and Antoine Girard. Hybridization methods for the analysis of nonlinear systems. *Acta Inf.*, 43(7):451–476, January 2007.

[2] M. Berz and K. Makino. Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. *Reliable Computing*, 4(4):361–369, 1998.

[3] X. Chen. *Computing rigorous bounds on the solution of an initial value problem for an ordinary differential equation*. PhD thesis, RWTH Aachen University, 2015.

[4] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Taylor model flowpipe construction for non-linear hybrid systems. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 183–192. IEEE, 2012.

[5] X. Chen and S. Sankaranarayanan. Decomposed reachability analysis for nonlinear systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 13–24, Nov 2016.

[6] A. Eggers, N. Ramdani, N. Nedialkov, and M. Fränzle. Improving SAT modulo ODE for hybrid systems analysis by combining different enclosure methods. In *Proc. of Intl. Conf. on Software Engineering and Formal Methods*, SEFM'11. Springer, 2011.

[4] Although these independent parts compute AND functions, they do not resemble the destiffened AND-Gate.

[7] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *Computer Aided Verification*, pages 379–395. Springer, 2011.

[8] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48(5):1038–1068, September 2001.

[9] S. Kong, S. Gao, W. Chen, and E. Clarke. dReach: δ-reachability analysis for hybrid systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015.

[10] M. R. Lakin, S. Youssef, F. Polo, S. Emmott, and A. Phillips. Visual DSD: a design and analysis tool for DNA strand displacement systems. *Bioinformatics*, 27(22):3211–3213, 2011. http://research.microsoft.com/dna.

[11] Y. Lin and M. A. Stadtherr. Validated solutions of initial value problems for parametric ODEs. *Appl. Numer. Math.*, 57(10):1145–1162, October 2007.

[12] R. J. Lohner. Enclosing the solutions of ordinary initial and boundary value problems. *Computer Arithmetic, Scientific Computation and Programming Languages*, pages 255–286, 1987.

[13] K. Makino. *Rigorous Analysis of Nonlinear Motion in Particle Accelerators*. Michigan State University. Department of Physics and Astronomy, 1998.

[14] K. Makino and M. Berz. Suppression of the wrapping effect by Taylor model-based verified integrators: Long-term stabilization by preconditioning. *International Journal of Differential Equations and Applications*, 10(4), 2005.

[15] R. E. Moore. Automatic local coordinate transformations to reduce the growth of error bounds in interval computation of solutions of ordinary differential equations. *Error in digital computation*, 2:103–140, 1965.

[16] N. S. Nedialkov. *Implementing a Rigorous ODE Solver Through Literate Programming*, pages 3–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[17] A. Platzer and J. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems (system description). In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, *Automated Reasoning*. Springer, 2008.

[18] P. Prabhakar and M. Viswanathan. A dynamic algorithm for approximate flow computations. In *Proc. Intl Conf. on Hybrid Systems: Computation and Control*, HSCC '11. ACM, 2011.

[19] R. Testylier and T. Dang. NLTOOLBOX: A library for reachability computation of nonlinear dynamical systems. In D. Van Hung and M. Ogawa, editors, *Automated Technology for Verification and Analysis*. Springer, 2013.