Decomposition of Strongly Coupled Systems

Dominik Bucher, Ilias Garnier, Ricardo Honorato, Vincent Danos*
University of Edinburgh, ETH Zurich

27 June 2013

For analysis and execution efficiency, many simulation models could benefit from decomposition into modules. We are doing research on methods that allow decomposition even for strongly coupled systems, i.e. systems in which certain variables influence all others. Furthermore, we analyze how to exploit a given modular decomposition by developing a technique of 'uchronic' execution where modules interpolate late values and can backtrack when actual values eventually reveal substantial discrepancies.

1 Introduction

Many simulation models, especially in biology, benefit from integrating different small modules into a large system. This is partly because the complexity of the overall system is immensely big, so that a single monolithic block would be difficult to describe and reason about, and also because single modules might use different modeling approaches, like ordinary differential equations, boolean networks, flux balance analysis and more. Of special interest are so-called whole cell models [5], which try to describe and simulate everything that happens within a biological cell. But also metabolic [7], protein [3] and other networks can be modularized for analysis and simulation.

Equally interesting and closely related, another approach tries to form smaller connected systems from a large monolithic block [1, 9]. Such decompositions are useful for detailed analysis of components as well as for efficient simulation if used in combination with other tricks. However, performing this task manually can be cumbersome, error-prone and time consuming. We are working on algorithms that are able to split and modularize arbitrary systems into smaller components which can be simulated on different timescales with synchronizations at optimal times. The principles of distributing variables among modules are formalized and analyzed.

The systems discussed are of a very general structure as outlined in section 2. Modules are built of communicating processes, which are to be seen as black boxes themselves. As a more accurate analysis is possible when concrete instances of processes are considered, we study detailed systems of ordinary differential equations (described in section 3). The study is done analytically and will be complemented by benchmarking various biological models. Section 4 talks about so-called uchronic systems which arise from modularization and allow processes to predict the outputs of other processes and thus advance further in time. As this allows easy rebalancing, it is of particular interest in environments where computing resources are scarce and computation load should be maximized. Section 5 concludes our current findings and further steps.

*In alphabetical order

2 Modularization of Systems

The problem of splitting a large system into smaller subsystems or so-called modules is of particular interest for huge models where efficiency can be greatly increased by letting single modules run in parallel, with synchronization rounds at certain times. An optimal splitting takes into account the maximization of the step sizes of different modules, an equal average computation load for all modules and the total number and size of modules. In a first step, we study a general class of systems built from two basic components. In a second step we look at interactions between those components.

2.1 A Ubiquitous Definition of Systems

A collection of *state variables* called state vector and several communicating *processes* form the two basic components. We will later introduce *resources* which are state variables that are shared between multiple modules.

A state variable $s_i \in \{s_1, ..., s_n\}$ is any modifiable object in the simulation (e.g. a number or a string). The complete set of state variables, also denoted as the *state vector* describes the whole simulation state (processes have no private state in this formalization) at time t:

$$S(t) = \begin{pmatrix} s_1(t) \\ s_2(t) \\ ... \\ s_n(t) \end{pmatrix} \tag{1}$$

A communicating *process* $P_i \in \{P_1, ..., P_p\}$ is a purely functional entity that produces a change C on the simulation system within a time period Δt starting from initial conditions $S(t)$ at a time t. With S denoting the space that resources span (e.g. $\mathbb{R}$ for real numbers or $\{0, 1\}$ for Booleans), a process is defined as:

$$\begin{aligned} &P : S^n \times \mathbb{R} \times \mathbb{R} \to S^n \\ &(S(t), t, \Delta t)) \mapsto C(S(t), t, \Delta t) \end{aligned} \tag{2}$$

A *change* C has the same structure as the set of resources $S(t) \in S^n$ and describes the change the process would like

to perform on the simulation state during $t + \Delta t$. The change a process calculates is the collection of all different smaller changes that happen on the simulation state during the execution of P.

2.2 Process Interactions

In any system variables will depend on other variables with different strength. With the above formalism this depends on the changes produced by the processes. The *dependency matrix* J describes the coupling of variables, and is defined for each process P_i:

$$J_i = \begin{bmatrix} \frac{\partial C_{s_1}}{\partial s_1} & \cdots & \frac{\partial C_{s_1}}{\partial s_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial C_{s_n}}{\partial s_1} & \cdots & \frac{\partial C_{s_n}}{\partial s_n} \end{bmatrix} \quad i \in 1, ..., p \tag{3}$$

Note the similarity to the *Jacobian* matrix. In particular, variable s_i depends on variable s_j if $J_{ij} \neq 0$. The *Jacobian* can be used for the case of ordinary differential equations, but in a general system, dependencies have to be determined in other ways, e.g. with some sensitivity analysis tailored to the modeling technique used.

The modularization algorithm uses the dependency matrix to cluster variables into - if desired, hierarchical - modules that can be simulated with maximal time steps, but computational loads still in the same range. Often, there are variables that influence many other variables strongly. If the algorithm were to try to modularize without taking such variables into account, the result would often be a monolithic block again. Thus we adapted the algorithm to detect these so-called *resources* and we are developing and analyzing techniques to share them between modules without additional communication. Those techniques revolve around additive splitting, sharing and reintegrating and allow to decouple a system as if no *resources* were existent. An exemplary sharing technique is based on momentarily fluxes $\phi_{i,j}$ (flux of process i in resource j), where resources are distributed proportional to the fluxes:

$$s_{i,j} = \alpha_{i,j} \cdot s_i \quad \text{with } \alpha_{i,j} = \frac{\phi_{i,j}}{\sum_i \phi_{i,j}}$$

3 An Exemplary System of Linear Differential Equations

This section outlines a modularization example, namely a system of differential equations. Differential equation systems have been studied to a great extent and they provide a good basis for comparison and benchmarking. We look at linear systems of the following form:

$$\begin{pmatrix} y_1' \\ y_2' \\ \vdots \\ y_m' \end{pmatrix} = \begin{pmatrix} F_1\left(t, y_1, y_2, \ldots, y_m\right) \\ F_2\left(t, y_1, y_2, \ldots, y_m\right) \\ \vdots \\ F_m\left(t, y_1, y_2, \ldots, y_m\right) \end{pmatrix} \tag{4}$$

Or also just $\mathbf{y}' = N \cdot \mathbf{y}$ in matrix form.

The system allows an easy computation of the dependency matrix J by calculating the *Jacobian* which is then used by the algorithm to modularize it. The modularization algorithm uses the dependency matrices to build sensitivity and influence vectors (where sensitivity denotes the overall sensitivity of a process to a given variable and influence denotes how fast a variable changes as its calculation inputs change). Using these matrices clusterings are found, either per variable or also for the whole state vector. The clusterings denote how frequent two processes have to communicate.

The modularization is now optimized by generating clusters that have to communicate as little as possible. The optimization algorithm is adapted from previously researched methods [6] and optimizes the network modularity

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - P_{ij}\right] \delta(\mathcal{M}_i, \mathcal{M}_j)$$

where A is calculated from J, n and m are vertices and edges, k_i is the number of edges connected to node i and

$$P_{ij} = \frac{k_i k_j}{2m}$$

$$\delta(\mathcal{M}_i, \mathcal{M}_j) = \begin{cases} 1 & \text{if nodes } i, j \text{ in same module} \\ 0 & \text{else} \end{cases}$$

In contrast to previously researched optimization algorithms [1, 9], we are examining additional mechanisms to handle highly coupled systems (where some variables are tightly bound to more than one module). For some mechanisms our work relies on the assumption that resources are sufficiently abundant to saturate the system so that modules consume them at a constant rate.

In order to be able to benchmark the modularized resulting system, we will perform the modularization on known systems from biology (where systems of linear ordinary differential equations are common, e.g. in metabolic networks, where N becomes the stoichiometry matrix) and look at the resulting performance of the system. Performance measures include total processing time, overall simulation error and comparison with algorithms that don't use any special notion for shared variables.

4 Uchronic Execution of Processes

In this section, so-called uchronic (coined from the French *uchronie*) execution of processes is discussed. Starting from a modularized system as generated above, modules are now allowed to predict outputs of other modules in order to let them advance faster in time. This is especially valuable if certain processes use much less CPU time and would thus always have to wait for others. Such processes now take advantage of their lower CPU load to advance further in (simulation) time. Once a certain time is reached, they only activate to check if their predictions were within an acceptable bounds and release CPU resources otherwise. In case their predictions were wrong, they have to backtrack and restart their simulation from the violating point in time. However, as modules are only loosely coupled, predictions

seem to be correct with high probability, an argument on which we we will do further research.

The design and the correctness proof of this whole backtracking mechanism should not be the responsability of the model designer, but should rather come as a generic layer on top of the model. In order to achieve that, we propose a process algebra equiped with an explicit notion of causality and an operational mean of consistently backtracking in a distributed way whenever some inconsistency is detected.

The next section starts with a few words on the formalization of reactive processes, is followed by an introduction of some notations related to the treatment of causality and the presentation of our process algebra.

4.1 Semantics of reactive processes

LTSs can equivalently be presented under their more abstract, coalgebraic guise. Where we had a set of states Q and a transition relation $\rightarrow \subseteq Q \times \mathcal{L} \times Q$ for some set of labels $\mathcal{L}$, we can more abstractly reason about the same set Q equipped with a so-called *observation function* $f : Q \rightarrow \mathcal{L} \times Q$. This kind of correspondence can be systematized and generalized greatly. Given any set function $F : Set \rightarrow Set$ specifying the observations, an $F - coalgebra$ is a pair $(X, f : X \rightarrow F(X))$, compactly representing a transition system. For a wide class of such set functions, there exists a so-called *final coalgebra*, which happens to be the canonical set of processes with the specified signature. We won't insist more on this particular subject in these lines. The interested reader can refer to [8, 2] for more information.

We assume that we are given a notion of *trace* which for any state $x_0 \in X$ of a deterministic coalgebraically presented transition system produces the possibly infinite execution sequence $x_0 \cdot x_1 \cdots$ of states.

4.2 The partial order of causality on activations

We use a notion of causality reminiscent of event structures [10]. Let E be the countably infinite set of events. A *partial order on events* is a tuple $\langle E, \leq \rangle$ where $\leq \subseteq E \times E$ is a partial order relation and such that for any $e \in E$, the set of causes $\mathcal{C}(e) \triangleq \{c \leq e \mid c \in E\}$ is finite. A given total (i.e. bijective) ordering of events $w \in E^{\mathbb{N}}$ is *correct* iff:

1. $\forall x < y \in \mathbb{N}, \neg(w_y \leq w_x)$, i.e. causality is respected,
2. $\forall y \in \mathbb{N}, \forall c \in \mathcal{C}(w_y), \exists x < y, w_x = c$, i.e. events are properly justified by their causes.

Such an ordering w is *partially correct* iff there exists an infinite subset of prefixes p of w for which there exists a correct permutation. Finally, it is *causal* when only the first condition is verified. These notions are trivially extended to traces given a function from states to events.

4.3 Uchronic processes

Let us consider a particular set of reactive processes: they can input and output data, and they act as causal stream functions [4]. In practice, their dynamics is embedded in real time: they sample their input at various times (possibly in a non-periodic way), updating simultaneously and in an instantaneous way their current output. Each process has its own activation schedule, which might possibly be dependent on input data and which has to be treated as being non-deterministic.

Rather than explicitly using a jacobian-like matrix, we model inter-process couplings by ascribing an abstract partial order on activations, *not* to be mistaken with the total order imposed by physical time. In practice, at any activation point, a process will either activate normally or *interpolate* its input data if the partial order on activations is not satisfied (meaning intuively that some required input data is not available). When an inconsistency is detected, a distributed backtracking mechanism is activated.

Let A and B be respectively the sets of inputs and outputs of a process. In order to encode distributed backtracking, we need to add some information on top of those sets. In essence, a "backtrack" signal should contain enough data for the receiving ends to go back to the last correct consistent activation, identified as a particular event. This defines our effective input and output sets $\overline{A}$ and $\overline{B}$ where:

$$\overline{X} = \{\mathsf{ok}(x, e) \mid x \in X, e \in E\} \cup \{\mathsf{backtrack}(e) \mid e \in E\}.$$

As can be guessed, the second component of the disjoint union signals backtracking.

We model a *process* as a dynamic and stateful entity interacting with an *environment* representing the rest of the system. The environment coalgebra is (X_e, obs_e) where the signature of the observation function is:

$$obs_e : X_e \rightarrow \overline{A} \times \wp_{fin}(E) \times \left(\overline{B} + 1 \rightarrow X_e\right).$$

Each activation is labelled by all events produced during that activation and that are necessary to justify the provided input. The coalgebra of processes is (X_p, obs_p), where the second component has signature:

$$obs_p : X_p \rightarrow E \times (X_e \times X_p)^* \times (\wp_{fin}(E) \rightharpoonup A \times E) \times \left(\overline{A} \rightarrow \overline{B} \times X_p\right).$$

We label each activation of a process with its unique event, a stack of states storing the past of the system for backtracking purposes *and* a mapping from missing causes to previously interpolated events. The environment acts as a provider of inputs and a consumer of outputs, whereas the process does the opposite. To simplify matters here, we allow processes to produce nothing. The system evolves by synchronously interleaving the successive unfoldings of the process and environment coalgebras. The state space of the system is $X_e \times X_p$.

Given these informations and some other constraints, we assert that it is possible to reason about the local causal consistency of a process activation. In this abstract, we restrict our attention to the case where activation events are only justified by their past and at most one external event. We note $Justified \subseteq (X_e \times X_p)$ the states where the activation of the process component is justified.

Our aim is to provide a uchronic layer above regular processes, for which we impose the type signature $S \rightarrow (A \rightarrow B \times S)$, assuming some state space S. It is thus necessary to *define* X_p generically in function of S. There indeed exists a function that given i) a finite set of regular processes, ii) a causal assignement of events to each process activations, iii) an interpolation method $interp : S \rightarrow A$,

iv) some symmetric consistency relation on inputs $Con \subseteq A \times A$ and *v*) a partially correct scheduling $w \in E^{\mathbb{N}}$ creates a consistent system. The semantics is defined by case analysis on the input and is sketched below.

I. $\mathsf{ok}(a, e)$ **case:**.

1. If the activation event of the process is *correctly* justified by an ok input, the process checks whether this event justifies a previous speculative activation.

- If it is the case and the interpolated input is consistent with the actual one according to Con then proceed as usual, if it is not consistent then bactrack;
- If the event does not justify a previous activation, treat it regularly.

2. If the activation event is *not* justified by the input, meaning that the required event has not occured and that we have to "guess" it by interpolation.

II. $\mathsf{backtrack}(e)$ **case:**

The processes computes its last state non causally dependent on the event e and invalidates all its outputs that are causally dependent on e.

In what follows, we give a formal account for a representative subset of the operational semantics. A consistent interpolation gives rise to the following rule:

$$\frac{\begin{array}{l} obs_e(x_e) = (\mathsf{ok}(a, e_a), evs, next_e) \\ obs_p(x_p) = (e, stack, mapping, next_p) \\ (x_e, x_p) \in Justified, mapping(\{e_a\}) = (a_i, e_i) \\ (a, a_i) \in Con, x'_e = next_e(\bullet) \end{array}}{(x_e, x_p) \to (x'_e, x_p)}$$

Where $\bullet$ is the element of the singleton set. Sending a backtrack message is taken care of by this rule:

$$\frac{\begin{array}{l} obs_e(x_e) = (\mathsf{ok}(a, e_a), evs, next_e) \\ obs_p(x_p) = (e, stack, mapping, next_p) \\ (x_e, x_p) \in Justified, mapping(\{e_a\}) = (a_i, e_i) \\ (a, a_i) \notin Con, x'_e = next_e(\mathsf{backtrack}(e_i)) \\ x'_p = stack(e_i) \end{array}}{(x_e, x_p) \to (x'_e, x'_p)}$$

where $stack(e_i)$ denotes the unique (by assumption) previous state of the process labelled with event e_i. Receiving a backtrack message is handled in the following way:

$$\frac{\begin{array}{l} obs_e(x_e) = (\mathsf{backtrack}(e), evs, next_e) \\ obs_p(x_p) = (e, stack, mapping, next_p) \\ x'_e = next_e(\mathsf{backtrack}(e_b)), x'_p = stack(e) \end{array}}{(x_e, x_p) \to (x'_e, x'_p)}$$

where e_b is the event corresponding to the most recent (and unique) activation of the process such that $e \not\leq e_b$. We omit the other rules, but they are in the same spirit.

As can be proved, the fact that this mechanism gives rise to consistent executions relies on the assumption that the actual scheduling imposed on the system is partially correct, meaning that delayed events eventually happen and that consistency is achieved. Other points of interest, such as compositionality, are not discussed here.

5 Conclusion

We described a general method allowing to automatically decompose hierarchically a system into communicating processes, given some way of assessing their degree of interdependence. Starting from such a decomposition, we can go further and decouple the actual dynamics of the processes thanks to the uchronic model of execution. Together, these elements give the foundation for a general software framework that promises efficiency and scalability. We are currently prototyping such a framework.

Some areas to investigate are the automatic extraction of jacobian-like matrices from black-box processes using sampling, stochastic uchronic dynamics and the possible application of machine learning techniques to smart interpolation.

References

[1] Michael Ederer, Thomas Sauter, Eric Bullinger, Ernst-Dieter Gilles, and Frank Allgöwer. An approach for dividing models of biological reaction networks into functional units. *Simulation*, 79(12):703–716, 2003.

[2] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.

[3] Hawoong Jeong, Sean P Mason, A-L Barabási, and Zoltan N Oltvai. Lethality and centrality in protein networks. *Nature*, 411(6833):41–42, 2001.

[4] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[5] Jonathan R Karr, Jayodita C Sanghvi, Derek N Macklin, Miriam V Gutschow, Jared M Jacobs, Benjamin Bolival, Nacyra Assad-Garcia, John I Glass, and Markus W Covert. A whole-cell computational model predicts phenotype from genotype. *Cell*, 150(2):389–401, 2012.

[6] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582, 2006.

[7] Erzsébet Ravasz, Anna Lisa Somera, Dale A Mongru, Zoltán N Oltvai, and A-L Barabási. Hierarchical organization of modularity in metabolic networks. *science*, 297(5586):1551–1555, 2002.

[8] J. J. M. M. Rutten. Universal coalgebra: a theory of systems, 2000.

[9] Julio Saez-Rodriguez, Stefan Gayer, Martin Ginkel, and Ernst Dieter Gilles. Automatic decomposition of kinetic models of signaling networks minimizing the retroactivity among modules. *Bioinformatics*, 24(16):i213–i219, 2008.

[10] Glynn Winskel. Event structures. In *Advances in Petri Nets*, pages 325–392, 1986.