Appendices

Appendix A: The automated model behavioural analysis framework (AMBA)

In designing the framework for automated model behavioural analysis (AMBA), the routines for integration, the model equations and the behavioural analysis were separated as much as possible. They are independent entities exchanging information, each having with their own responsibilities. This modular and generic set up allows us to replace and change any one of the three routines without affecting the others. The three parts of the framework for automated model behavioural analysis exchange information as depicted in Figure 1.

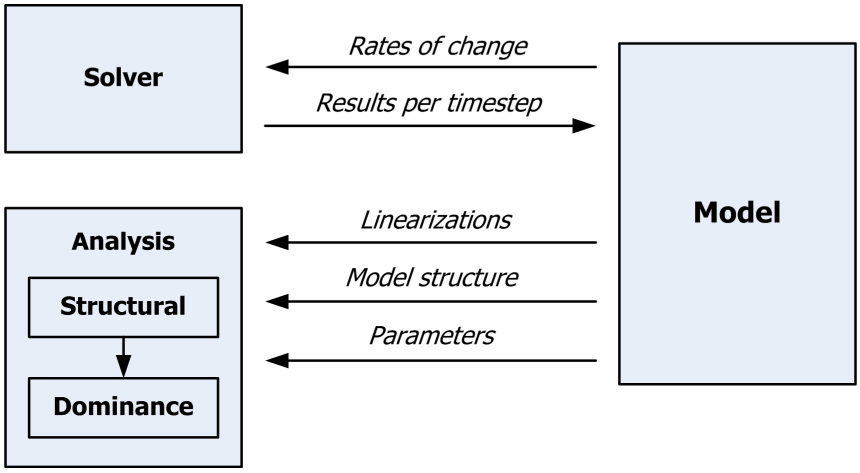


Figure 1: Information exchanged in the framework for Automated Model Behavioural Analysis

Appendix A.1:Main procedure

The main procedure executes model runs and analyses; linking the separate parts of the AMBA framework together. It calls the functions required for the structural and dominance analyses of the model, while using the solver to run the model. The procedure needs the following inputs: (i) a location reference to the model in a representation suitable for use within the framework, (ii)a vector of time steps for the integrator, (iii) a separate vector of times at which to perform the analysis, and (iv) a location reference to the solver to be used.

After performing an initial structural analysis, the procedure runs the model until the first moment at which a full behavioural analysis is required. This analysis is performed, the results stored and the interrupted run is continued until the next moment at which full behavioural analysis is required. This continues until the last snapshot time for behavioural analysis has been reached. The Matlab Code of the main procedure is included in Appendix A. For an overview see Figure 2.

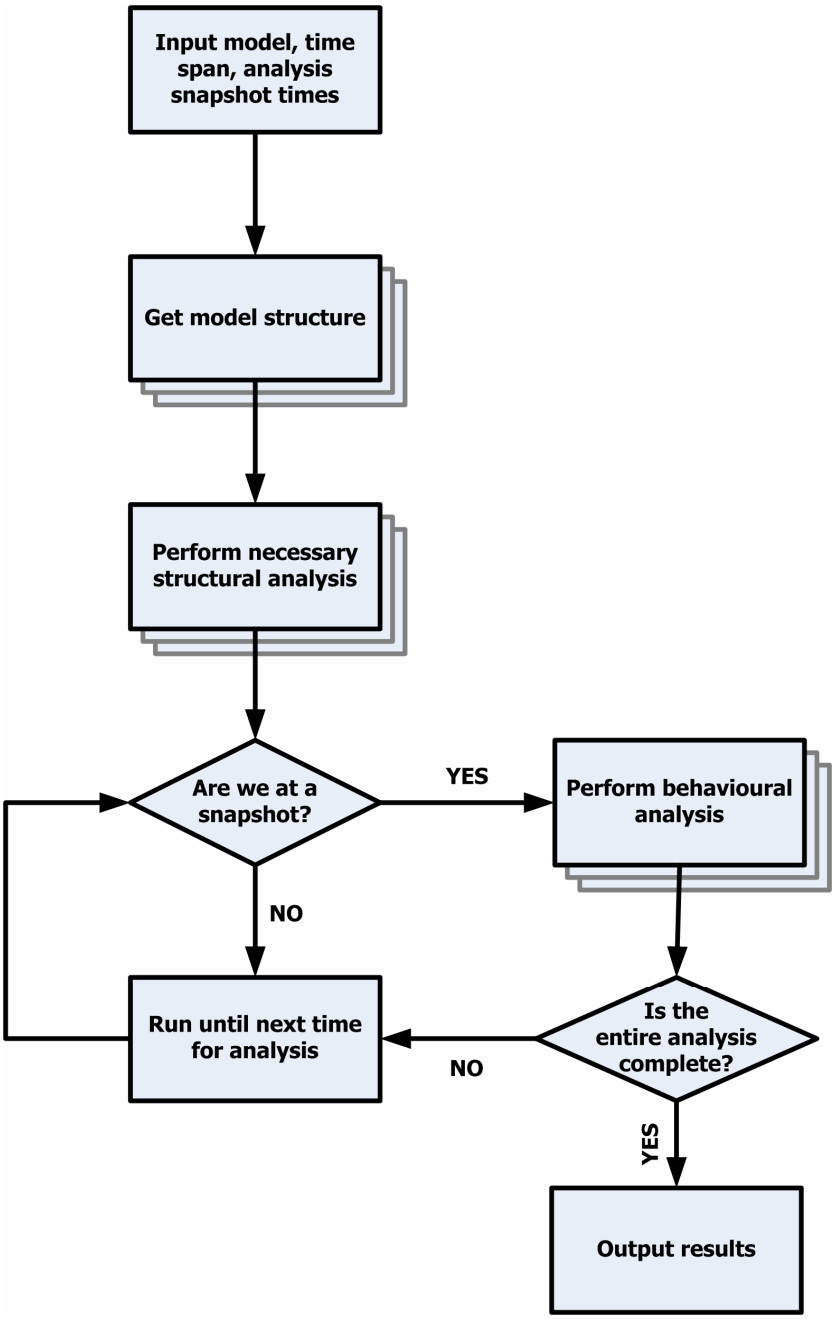


Figure 2: Flowchart of the main procedure of the AMBA framework. The structural analysis uses the algorithms defined by (Oliva 2004) . The block "perform behavioural analysis" executes the model behavioural analysis at a specific point in time.

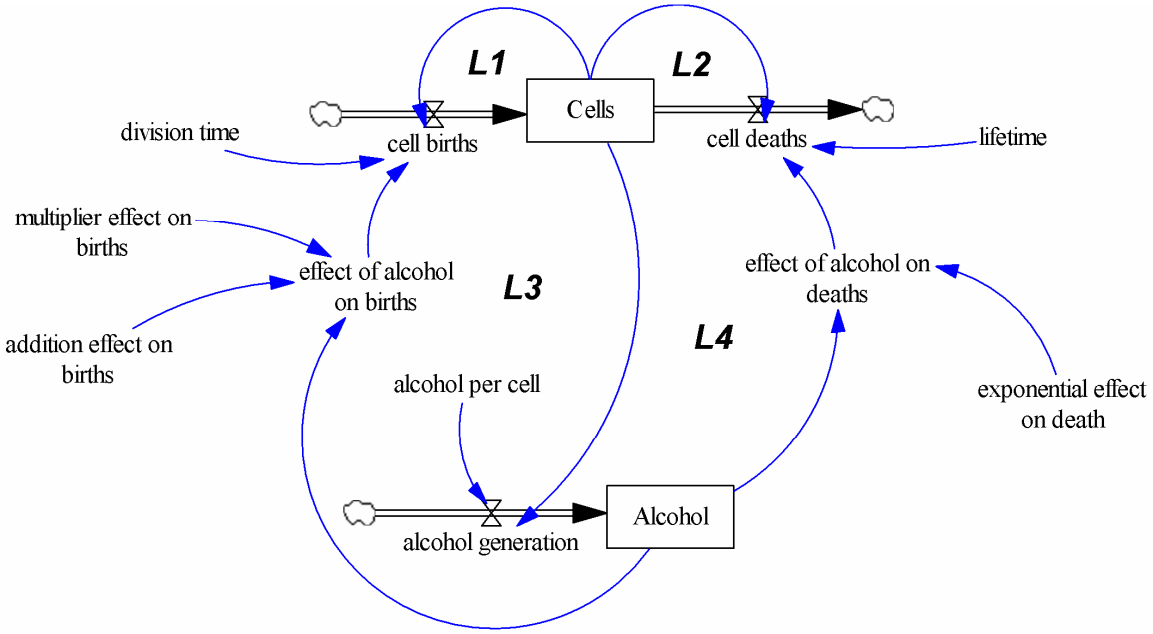
Appendix A.2:Model Representation

The model is designed to keep all unnecessary complexity away from the other AMBA components. It is an outer shell, hiding its internal workings from the rest of the framework. It supplies all the information needed to run and analyse the behaviour of the model. This includes linearizing and building adjacency matrices (Oliva 2004) as well as calculating the net rates of change of state variables. At this stage, edge gains for the linearizations are calculated using simple finite differences, but these can be calculated

analytically on a per variable basis should this prove necessary. To summarize, the model:

- Receives the results of the integration for each time step from the integrator and in turn provides the net rates of change of the state variables to the integrator.
- Provides the behavioural analysis routine with a graph representation (an adjacency matrix, for instance) of the model so that structural analysis can be performed.
- Provides the behavioural analysis routine with the information needed to perform the dominance analyses.

Appendix B: Yeast Model



Appendix B.1: System Structure

Table 1: Directed Cycle Matrix of the Yeast Model

Edge		Loop			
From	**To**	**L1**	**L2**	**L3**	**L4**
Cells	Births	1	0	0	0
Births	Cells	1	0	1	0
Cells	Deaths	0	1	0	0
Deaths	Cells	0	1	0	1
Cells	Alcohol Generation	0	0	1	1
Alcohol	Alcohol	0	0	1	1

Generation					
Alcohol	EffAlcBirth	0	0	1	0
EffAlcBirth	Births	0	0	1	0
Alcohol	EffAlcDeath	0	0	0	1
EffAlcDeath	Deaths	0	0	0	1

Appendix B.2: Properties of the gain matrix of the Yeast model

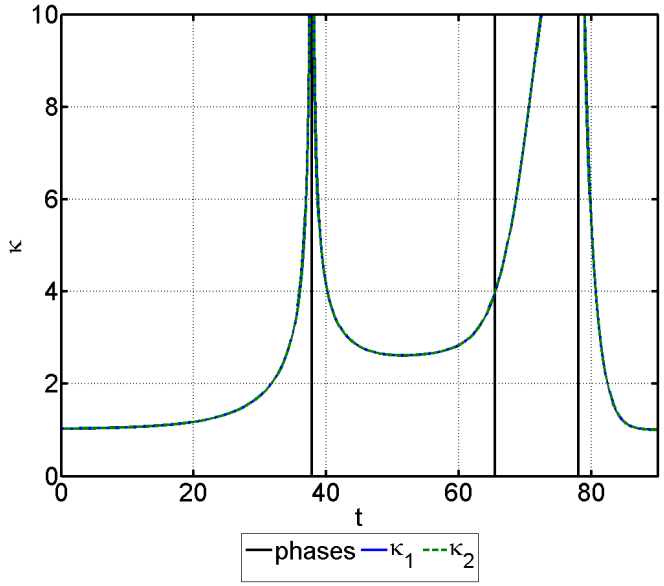


Figure 3: Condition number of the eigenvalues of the Yeast model

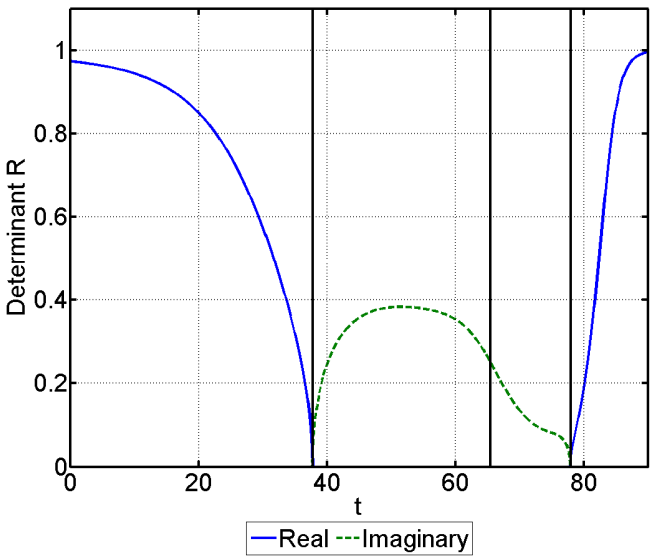


Figure 4: Determinant of the matrix of right eigenvectors of the Yeast model

Appendix C: Matlab code

Appendix C.1: The main analysis procedure

```
function [realRes, imagRes, overRes, eigenRes, t, y, condires, auxvals]...
    = automagicTestK(theModel, timespan, snapShots, integrator)

    % Model made global for wrapper function when integrating. simpleInteg
    % just wraps a matlab function around the java model making its easier
    % to integrate
    global model

    % Java imports
    import nl.tudelft.tbm.pa.rubberband.util.*;
    import nl.tudelft.tbm.pa.rubberband.*;


    % Model reference
    model       = theModel;

    % Assuming constant structure, do the structural analysis first.
    % Do not include model parameters
    adjac       =...
        model.getAdjacencyMatrix(AdjacencyMatrix.EXCLUDE_PARAMETERS);
    % Get the DCM and related edges. Requires translation from output by
    % oliva.
    [edges DCM] = getSilsDCM(adjac);

    % Determine the end of time
    endOfTime   = timespan(numel(timespan));

    % Initialize vectors for results of integretation
    y = zeros(1, model.getStates().length);
    t = 0;

    % Initialize contros for time and snapshots
    snapIndex       = 1;
    tLast           = timespan(1);
    deltat          = .1;
    % Time at which to analyze
    snapTime        = snapShots(snapIndex)
    remainingTime   = timespan;
    finished        = 0;

    % Run while taking stopping the model at  for analyses.
    % Analyse model snapshots in separate (nested) function.
    while ~finished
        % Get the snapshot, else if (time < snapshot) run till snapshot
        if tLast == snapTime
            % Analyze and store results in a 3d result matrix
            % Call the Kampmann analysis functions
            [realR, imagR, eigenss, condi] =...
                analyzeKampmann(model, edges, DCM);
            % Rescale the results for readability
            % Real part of the elasticity
            realRes(:,:,snapIndex) = reScale(realR,1);

            % Imaginary part of the elasticity
            if nnz(imagR) > 0
                imagRes(:,:,snapIndex) = reScale(imagR,1);
            else
                imagRes(:,:,snapIndex) = zeros(size(imagR));
            end
```

```
            % Overall elasticity
            overRes(:,:,snapIndex) = ...
                reScale(abs(complex(realR, imagR)), 1);
            % Eigenvalues of the model
            eigenRes(:,snapIndex)  = eigenss;
            % condition number of the eigenvalues
            condires(:,snapIndex)  = condi;

            snapIndex   = snapIndex +1;
            if snapIndex > length(snapShots) | snapTime == endOfTime
                finished = 1;
            else
                snapTime = snapShots(snapIndex);
            end
        else
            % Run till snapshot
            % Determine the vector of the timespan until the next snapshot
            % first.
            idx = find(remainingTime >= snapTime, 1, 'first');
            tidx = remainingTime(idx);
            % If snaptime falls exactly on a timestep
            if tidx == snapTime;
                integVector     = remainingTime(1:idx);
                remainingTime   = remainingTime(idx:numel(remainingTime));
            % Else if it falls between timesteps
            elseif tidx > snapTime
                integVector     = [remainingTime(1:idx-1) snapTime];
                remainingTime   =...
                    [snapTime, remainingTime(idx:numel(remainingTime))];
            end
            states = model.getStates();
            for statesIdx = 1:length(states)
                inits(1, statesIdx) = states(statesIdx).getValue();
            end
            % Integrate. Integrator is a function handle, which
            % should cover the independency.
            [ts ys] = integrator(@simpleInteg, integVector, inits);
            % Vertcat the solution, do not include redundant steps.
            t = vertcat(t, ts(2:length(ys)));
            y = vertcat(y, ys(2:size(ys,1),:));
            % Determine the current time of the model
            tLast = integVector(numel(integVector));
        end
    end

    % Remove dummy rows from result matrices
    t(1,:) = [];
    y(1,:) = [];
end
```

Appendix C.2: Code for the modified version of evaluating the contribution

```
function varargout = eigenValueContribution(model, deltat, t)

    % This is a rewrite of EvCont, slightly modified to show testresults.
    % model : A reference to the system,
    % deltat : the deltat used to calculate the change in slope
    % (independent from integration and analysis times)
    % t      : current time in the model
    % x      : the current state of the system

    % Get the current rates of change (slopevector) from the model
    slps = model.getNetRates(t);
    % Get the gain matrix from the model
    gainMatrix = desc2jacob(model.getDescriptor(0));

    % Calculate linearized slopes
    slopes_zero     = slopesAtT(slps, gainMatrix, 0);
    slopes_delta    = slopesAtT(slps, gainMatrix, deltat);

    % Get the difference between t(0) and t_delta. Relative to deltat
    slopechange = (slopes_delta - slopes_zero)/deltat;

    % Reduction to real seems to work, the imaginary
    % part of the contribution of the eigenvalues always
    % occurs in conjugates, making the net
    % slopechange completely real. So, since that net effect is zero,
    slopechange = real(slopechange);

    function scs = slopesAtT(slps, gains, dt)

        % Vectors and eigenvalues all complex form, scs always result in
        % complex conjugates.
        [vectors lambdas] = eig(gainMatrix);

        % The fundamental matrix.
        % (Boyce and DiPrima 1996)
        psi     = vectors*(exp(lambdas*dt).*eye(size(lambdas)));
        % Calculate alphas based on current slopes. Alphas can be complex
        alphas  = inv(vectors)*slps;
        % Diagonalize alphas
        alphas  = diag(alphas);
        % calculate slope components, retain in 2d form to conserve
        % information about different lambdas
        scs     = psi*alphas;
    end

    varargout{1}    = slopechange;
    % As said before, the imaginary part of the actual slope can be
    % ignored. Net effect is zero due to being complex conjugates.
    % used to compare with numericallly solved system to verify method
    varargout{2}    = real(sum(slopes_delta, 2))';
    % Eigenvalues, used for another consistency check
    varargout{3}    = diag(lambdas);

end
```