# An Explicit Method for Decoupled Distributed Solvers in an Equation-Based Modelling Language

Robert Braun
Linköping University
Division of Fluid and Mechatronic Systems
Department of Management and Engineering
Linköping, Sweden
robert.braun@liu.se

Petter Krus
Linköping University
Division of Fluid and Mechatronic Systems
Department of Management and Engineering
Linköping, Sweden
petter.krus@liu.se

## ABSTRACT

The Modelica language offers an intuitive way to create object-oriented models. This makes it natural also to use an object-oriented solver, where each sub-model solves its own equations. Doing so is possible only if sub-models can be made independent from the rest of the model. One way to achieve this is to use distributed solvers by separating sub-models with transmission line elements. This offers robust and predictable simulations, simplified model debugging and natural parallelism. It also makes it possible to use different time steps and solver algorithms in different parts of the model to achieve an optimal trade-off between performance and accuracy. The suggested method has been implemented in the Hopsan simulation environment. Different modelling techniques for taking advantage of the distributed solver approach are explained. Finally, three example models are used to demonstrate the method.

## Categories and Subject Descriptors

I.6.8 [**Simulation and Modelling**]: Types of Simulation—*Distributed*

## General Terms

Theory

## Keywords

Keywords: distributed solvers; transmission line element method; Modelica; model generation

*EOOLT 2014,* October 10 2014, Berlin, Germany
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2953-8/14/10 ...$15.00
http://dx.doi.org/10.1145/2666202.2666212.

## 1. INTRODUCTION

When working with object-oriented simulation models, a logical approach would be also to use an object-oriented equation solver that resembles the object hierarchy in the model. Most state-of-the-art simulation environments, however, use centralized solver algorithms. While centralized solvers are often required for solving stiff equation systems, distributed solvers provide several advantages. It can for instance facilitate model debugging, increase mathematical robustness and enable parallel execution on multi-core platforms. This paper demonstrates how an equation-based language can be used in combination with the transmission line element method (TLM) in a simulation tool using distributed solvers. Individual parts of the model can then be made independent by introducing physically motivated time delays. Dividing a model in this way, where different sub-models exchange data at a fixed time interval, closely resembles the principles of a fine-grained co-simulation. In this paper, a general interface for using the Modelica language in combination with distributed solvers has been implemented in the Hopsan simulation environment. Users can arbitrarily mix equation-based models with TLM elements to create an optimal model structure. This is a continuation of preliminary work presented in [1].

### 1.1 Related Work

Some previous experiments to combine equation-based languages and TLM have been conducted. Methods for generating source code for distributed solvers have been demonstrated by Krus [2] using Mathematica and Johansson and Krus [3] using the Modelica language. The generated sub-models were subsequently connected using TLM in the Hopsan simulation tool. A more general method for connecting models from different simulation tools using the Functional Mock-up interface (FMI) [4] was presented by Braun and Krus [5]. Nyström and Fritzson [6] used TLM elements for parallel simulation of Modelica models on cluster computers. Automatic approaches utilizing model structure was recently presented by Casella [7] and Sjölund et al [8], where TLM elements were inserted by identifying weak couplings in the Jacobian matrix. Related work on partitioning Modelica models using methods similar to TLM have been conducted by Papadopoulos and Leva [9] and by Khaled et al [10].

The implementation in this paper differs from these method in that the user is free to mix pre-compiled models, equation based models and TLM elements in an intuitive, graphical interface. This provides a better reusability and flexibility. The modular solver structure is also suitable for multi-disciplinary simulations, co-simulations or real-time simulators. Most important of all, however, is that the model is never modified without an explicit action from the user.

### 1.2 Modelica

Modelica is an equation based object-oriented modelling language, developed by the Modelica Association [11]. Equations in Modelica describe pure equalities, and have no predefined causality [12]. They can therefore be written in any form, without specifying input or output variables. There is also support for algorithm sections, which contain assignments with specified output variables. Modelica is object-oriented and support class inheritance, which provide good reusability of models. The Modelica Association also develops the Modelica Standard Library, which provides predefined models for various domains.

## 2. DISTRIBUTED SYSTEM MODELLING

Most physical system can be modelled as a set of components with individual inductances (L) and capacitances (C), representing inertia and compressibility, respectively. For a fluid power system, these are defined by equations 1 and 2. Variables $p$ and $q$ represent pressure and flow, and can be replaced with the corresponding intensity and flow variables for any other physical domain.

$$q = C\frac{dp}{dt} \tag{1}$$

$$\Delta p = L\frac{dq}{dt} \tag{2}$$

Each component will have a time delay, which can be calculated from C and L. If the model is simulated with a time step of different size than its physical time delay, it is thus not possible to have physically correct values for both C and L at the same time. The state-of-the-art solution for this is to move either all inductance or all capacitance to a neighbouring component, so that each component only has either capacitance or inductance. In this way, the time delay can be either zero, or be assigned an arbitrary value by delaying the variables explicitly. With the transmission line element method (TLM), on the contrary, it is assumed that for components with very small inductance, for example weak springs or large hydraulic volumes, the error in the inductance will usually be insignificant for the simulation results. If the simulation time step is not very much larger than the corresponding time delays in the model, it is thus possible to decouple sub-components by replacing capacitive components with physically motivated time delays, see figure 1. This evolves naturally from calculating wave propagation, which occurs in all physical systems. Depending on the physical domain, waves can for instance consist of mechanical forces, electrical voltage or hydraulic pressure. A good overview of TLM is given by de Cogan et al [13].

The method originates from the method of bi-lateral delay lines [14] and from transmission line modelling [15]. It was first used in the HYTRAN simulation tool [16]. First parallel simulation was performed in 1991 using parallel computers [17]. Several experiments were later conducted using transputer technologies [18][19][20] and multi-core processors [21]. It is currently used in Hopsan, an object-oriented system simulation tool based on precompiled distributed solver components, and in the BEAST co-simulation package by SKF [22].

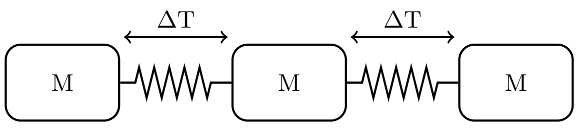

**Figure 1: Capacitances are treated as time delays, which can be used to decouple sub-models.**

The equations for a TLM element can be derived from the general solution of the Telegrapher's equations, as shown in equations 3 and 4.

$$\frac{\partial p}{\partial x} = -L\frac{\partial q}{\partial t} \tag{3}$$

$$\frac{\partial q}{\partial x} = -C\frac{\partial p}{\partial t} \tag{4}$$

Here $p$ and $q$ represent hydraulic pressure and flow. $C$ and $L$ are the capacitance and inductance of the element, respectively. The resulting equations for the two ends of a hydraulic pipe are shown in equations 5, 6, 7 and 8.

$$c_1(t) = Z_c q_2(t-\Delta t) + p_2(t-\Delta t) \tag{5}$$

$$c_2(t) = Z_c q_1(t-\Delta t) + p_1(t-\Delta t) \tag{6}$$

$$p_1(t) = Z_c q_1(t) + c_1(t) \tag{7}$$

$$p_2(t) = Z_c q_2(t) + c_2(t) \tag{8}$$

Because the element is assumed symmetric, the equations for both sides will be the same. The pressure $p$ is calculated as the sum of a wave variable $c$, representing the wave arriving from the other end of the connection, and the characteristic impedance of the element $Z_c$ multiplied with the flow $q$ at this end. $Z_c$ is the relationship between pressure and flow in the frequency domain and can be calculated from the capacitance and the inductance according to equation 9.

$$Z_c = \sqrt{\frac{L}{C}} \tag{9}$$

The time delay introduced between sub-models represents an actual time delay in the physical system. Consequently, the size of the delay becomes a parameter for the model. Any errors or instabilities caused by too large or too small delays can therefore be considered modelling errors, and not numerical errors. Thus, TLM is a numerically stable method. Due to the distributed solver approach, it is possible to use individual time-steps in each TLM element. Usually, however, the errors will be insignificant even when using a global time-step for all elements.

Separating a large system model into smaller independent models has several advantages. First, the behaviour of the model will be more predictive. Each sub-model will behave in the same way, no matter which system it is put into. Debugging the model also becomes much easier, because problems can be isolated to individual sub-models. Equations never need to be merged into a single system as with a centralized solver. In this way, the model hierarchy can be left intact. Models also become more robust against modifications, as a single sub-model cannot affect the numerical stability of the entire system [23]. Total simulation time also becomes the sum of the simulation time for each sub-model. For sub-models with fixed workload, simulation time will thus increase linearly with the size of the model. Another advantage is that the independence between sub-models makes it easy to run models in parallel on multi-core processors [21]. The fixed communication interval on the system level also makes the method suitable for real-time simulations [24].

## 3. IMPLEMENTATION

The suggested method has been implemented in the Hopsan simulation tool. To the authors knowledge this is the only available general-purpose system simulation tool with native support for TLM. The concept of distributed pre-compiled solvers makes it very suitable for code generation from for example equation-based languages.

## 3.1 Hopsan simulation tool

Hopsan is a multi-domain system simulation tool developed at Linköpings University. The first steps were taken in 1977, and in 1991 the transmission line element method was introduced [25]. In 2009 work began on a new version of the program, which is open-source, cross-platform and written in C++ [26]. Models in Hopsan consist of distributed sub-models connected with transmission line elements. Each sub-model is responsible for solving its own equations. This makes the sub-model naturally decoupled, so that their solvers can be executed concurrently. Previous experiments have shown linear speed-up when running large models on multi-core processors [21].

Hopsan models consist of two groups of components labelled Q-type and C-type, representing sub-models and TLM elements, respectively. These groups are solved in turns, so that all components of the same type solve their equations in parallel. Between each group is solved, the components interchange intensity, flow, wave and impedance variables through their connections. Q-type components compute intensity and flow, while the C-type components compute wave variables and impedance.

## 3.2 SymHop Symbolic Algebra Library

A code library for symbolic algebra was required to handle the mathematical operations. In order for it to work with Hopsan, it had to be cross-platform, support symbolic differentiation and have a compatible license. In the first attempts, the SymPy library written in Python was used [27]. It did, however, show poor reliability and unacceptably poor performance for this particular field of application [1]. Unfortunately, no other decent libraries that fulfilled the demands could be found. Instead, a customized C++ library called *SymHop* was created. In this way, the number of external dependencies could also be reduced.

Mathematical expressions are modelled as a tree structure consisting of *expression* objects. Each expression contains lists for different types of sub-expressions, such as terms, factors and divisors, see figure 2. The contents of these lists decide the type of the expression. An expression with terms, for example, is an addition. It is thus not allowed to have both terms and factors in the same expression. This approach makes trivial simplifications very easy to conduct; if a term contains its own sub-terms, these can easily be moved to the parent expression so that $(a+b)+c$ is simplified into $a+b+c$.

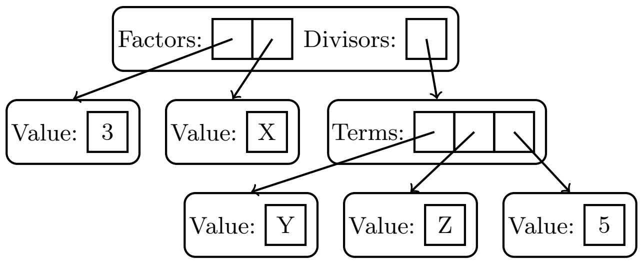

**Figure 2: SymHop tree describing the expression $3X/(Y+Z+5)$.**

The SymHop library greatly outperforms SymPy, mainly because the latter is written in an interpreting language while SymHop is pre-compiled. It is also designed with multi-threading in mind, for example by using re-entrant functions. The Modelica generator described below also uses some concurrent operations, mostly to differentiate the elements in the Jacobian matrix.

## 3.3 Model Generation From Modelica

Modelica code is loaded from external files, and can be modified from within Hopsan. These files specify the available Modelica models, which can in turn be used as components in the graphical interface. From the user perspective, there is no difference between a Modelica component and a pre-compiled component. It is possible to add Modelica components, pre-compiled components and TLM elements to the model and connect them in any order, as long as the connector types match. This means, for example, that a spring modelled in Modelica can easily be replaced by a TLM element without any other modifications to the model.

### *3.3.1 Distributed Solver Structure*

Hopsan uses distributed components with built-in solvers written in C++ [25]. Each component is compiled to a shared library file (`.dll` on Windows or `.so` on Linux) and dynamically linked by the main application at run-time. Components are then separated by TLM elements in the model and solved independently during the simulation. Everything that is to be simulated in Hopsan must thus be translated to this form. Each component consists of its own class, which inherits the `Component` base class. It is then registered in a factory class, from where objects of the specific component class can be created by using their unique type name. This makes it possible to reuse a component type in an unlimited number of instances. Usually all components are pre-compiled. Equation-based models, however, require a compilation step prior to the simulation. The first step is to identify independent groups of equation-based components, i.e. groups that are numerically isolated from the rest of the model by TLM elements. Each such group is then flattened into one single equation system.

The interface between Modelica components and the TLM elements are specified by using domain-specific connectors called `NodeHydraulic`, `NodeMechanic` and so on. When connecting two Modelica components to each other, such connectors behave as normal Modelica connectors. An example code for the NodeHydraulic connector is shown in listing 1. If connected to a TLM element, however, the code generator will replace it with a TLM port with input and output signals. Input signals consist of wave variable $c$ and characteristic impedance $Z_c$, and output signals of intensity and flow variables for the specified physical domain. $Z_c$ and $c$ are then used to compute the intensity according to equations 7 and 8. Practically, intensity and flow can thus be considered input and output variables, respectively.

```
connector NodeHydraulic "Hydraulic Node"
    Real        p "Pressure";
    flow Real   q "Volume flow";
end NodeHydraulic;
```

**Listing 1: Domain-specific connectors are used to interface Modelica models with TLM elements.**

For each connection to a TLM element, an additional equation is added, similar to equation 7 and 8. This is required for translating wave variable and characteristic impedance to intensity. In the second step, the Modelica code from each group is translated to compilable C++ code, which is subsequently compiled to shared library files. These file can then be linked against the Hopsan simulation core during runtime and thereby be used in Hopsan models. Each group of equation-based components is thus converted to one C++-component. Finally, a temporary model is created, to where the generated components are added. This model is only used for the simulation and never visible to the user.

This is illustrated by figure 3, representing a hydraulic position servo. The pump and the mechanical system are modelled in Modelica, while the rest of the hydraulic circuit is modelled using TLM. The mechanical parts and the pump are numerically isolated from each other, and can therefore be translated to two independent TLM components.

### 3.3.2 *Equation System Solvers*

Technically, any state-of-the-art numerical solver can be used to solve the DAE system. A great advantage with distributed solvers is the possibility to use different solvers in different parts of the model. In the current implementation, it is possible to choose between numerical integration

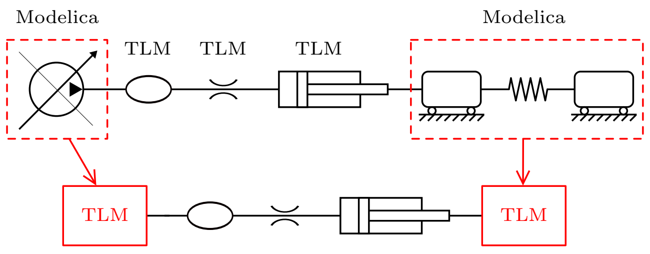

**Figure 3: Independent groups of Modelica components are identified and compiled to separate sub-models.**

solvers and a bilinear transform solver, where derivatives are replaced by transforms which are inserted directly into the equations, as described in [2]. Two implicit and four explicit Runge-Kutta methods for integration are available, ranging from first to fifth order. The trapezoid method is used as default, as it is an implicit solver with an infinite stability range. The algebraic equations are solved by transforming the Jacobian matrix using LU decomposition and subsequently solving it with a Newton-Raphson algorithm. Accuracy can be improved by solving the equation system iteratively in several steps. The number of iterations can be chosen from the user interface. A higher number will result in better accuracy at the expense of longer simulation time. For implicit methods, it is also possible to specify a tolerance limit and maximum number of iterations.

Another possible feature not yet included is variable time-step solvers. The real time application has been considered important, and for that situation, it is less useful. Even though the actual simulation loop uses a fixed communication time interval, each sub-model can still have its individual time-step. Using different time steps for different parts of the model can have a performance gain at the some order of magnitude as variable time-steps in time. Using both individual and variable time-steps in sub-models would make it possible to fine-tune the model for optimal performance. Taking advantage of these possibilities in an efficient way, however, remains a future challenge and has not been included in this paper.

The solving is performed by an equation solver utility class, which communicates with the sub-models, see figure 4. It is initialized with pointers to the state variables in the components and provides a *solve()* function, which performs the numerical integration steps for the specified solver. In return, the component must provide functions for calculating the time derivatives of the state variables, re-initializing variables and solving the algebraic equations. This is required in order to perform the iterations in the solver methods. The bilinear transform method does not require any integration and can thus solve the equation system directly.

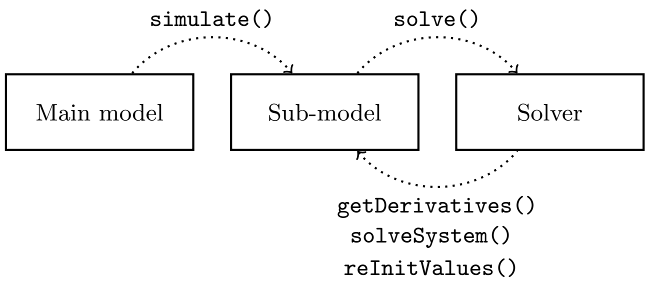

**Figure 4: A solver utility class, which communicates with the sub-model, is used to solve the equations.**

Benchmarking of the generated models shows that the major bottleneck is the LU decomposition of the Jacobian matrix. The time required for this increases quadratically to the number of coupled equations. It is thus very important to keep this number at a minimum level. For this purpose, a block lower triangularization (BLT) transformation is used.

## 4. MODELLING TECHNIQUES

By definition, the term *TLM model* can refer to any model where at least one capacitance is replaced with a TLM element. A more commonly used definition, however, is that it refers to a model where *all* capacitances are replaced by TLM elements. Any component in such a model is defined as a *TLM component*, which can be either one of the TLM elements or one of the components they are connected between. The latter definition is used below.

It is not necessary to simulate a TLM element with a time step exactly matching the corresponding physical time delay. If a larger time step is used, the inductance in the element will become too large. This is known as *parasitic inductance*. If the time step is sufficiently small, this will be insignificant compared to the inductance in the neighbouring components. This means that an element representing a small capacitance connected between small inductances will require a smaller time step. Thus, the time step of the model will be limited by the stiffest TLM connection.

One useful technique for TLM is thus to separate an existing model by introducing TLM elements only at the weak couplings. This is illustrated in figure 5, where a Modelica model is divided into two decoupled sub-models. In this way, the model can be partitioned into a course-grained distributed model, which can greatly improve performance.

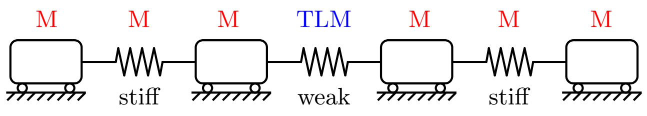

**Figure 5: By replacing weak connections in a Modelica model with TLM element, different parts of the model can be executed concurrently.**

It is also possible to approach the problem from the other direction, and introduce equation-based sub-models in a pure TLM model, as shown in figure 6. TLM models are often used for systems with significant capacitances, such as hydraulic or pneumatic circuits. Maximum size of the time step, however, is then limited by the smallest capacitance in the model. If there are only a few stiff connections in the model, these parts can be replaced by Modelica sub-models, while the rest of the model can still make use of the advantages of TLM.

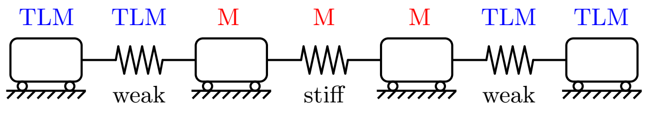

**Figure 6: If stiff connections in a TLM model is replaced by Modelica components, a larger time step can be used.**

In general, different physical domains, such as mechanics, hydraulics or electrics, have time delays of different magnitudes. A third approach could thus be to model different domains in the model with different modelling methods. The stiff physical domains can for example be modelled using Modelica, while the weaker domains are split up using TLM elements.

An alternative approach to the suggested methods above is to use TLM models with *distributed time steps*. Each part of the model can then have a time step that roughly resembles its physical time delay. This is, however, an unexplored field and more research needs to be done to prove its feasibility. A principal sketch of this is shown in figure 7.

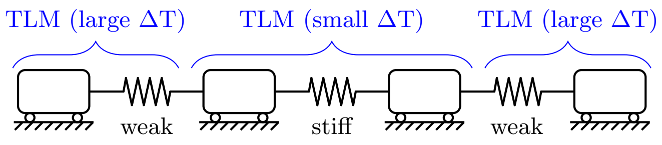

**Figure 7: An alternative solution can be to use TLM with distributed time steps.**

## 5. EXAMPLE MODELS

As mentioned above, one way of taking advantage of TLM is to replace weak couplings in an equation-based model with TLM elements. To demonstrate this method, a quite obvious example model is used, consisting of a car towing another car. In this model, the towing rope is modelled as a spring with slack, and has a very large capacitance compared to the rest of the model. The first car is driven by a simple engine model, both cars have wind resistance, and each wheel has an individual rolling resistance. First, the cars are accelerated to about 33 km/h. After 35 seconds, they hit a bump on the road. Results are shown in figure 9. The velocities for the first and second car are shown as solid and dotted lines, respectively.

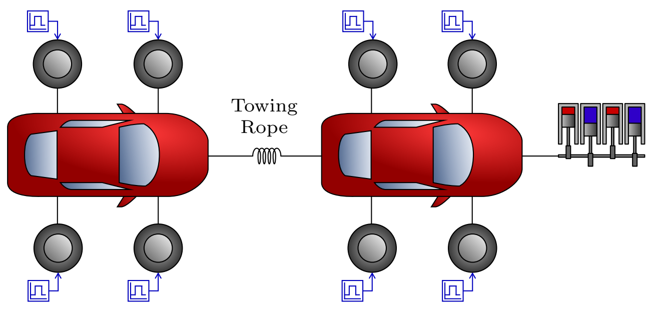

**Figure 8: A model of one car towing another car was used to demonstrate the method. The towing rope is replaced by a transmission line element.**

The results with and without the TLM element were indistinguishable. Simulation time with the TLM element was, however, 3.6 times faster. This has two reasons. First, a computer with a multi-core processor was used. This can, however, give a speed-up of at most 2, since the model was only divided into two sub-models. When simulation was limited to one thread, the remaining speed-up was 2.2. This can be explained by a reduction in the time required to solve the equation system numerically. In most cases the time required for solving an equation system increases superlinearly to the number of equations. In this case a simple solver using LU-decomposition were used, with a quadratic relationship between workload and number of equations. It should be pointed out that with a more efficient equation solver the speed-up is expected to be smaller, although still greater than 1.

Modelica code for the main model is shown in Appendix A. Annotations have been left out for readability. Note that no special language constructs are required. Hopsan will however use the prefix ”TLM_” as an identifier for non-Modelica components. This code is subsequently parsed in order to identify decoupled parts of Modelica code. In this cas,e there will be two such parts, one for each car. Generated code for the left car is shown in Appendix B. Hopsan has automatically added the TLM equations, marked by a red box.

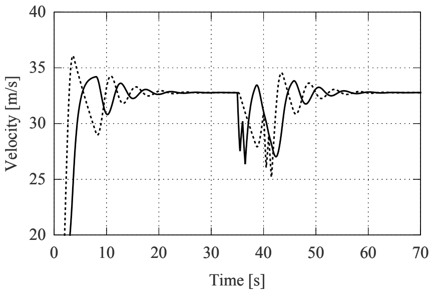

**Figure 9: Velocity of the first and second car in the model shown in figure 8. Simulation time is 3.6 times shorter after introducing the transmission line element.**

The second method mentioned in section 4 is analysed by a model of two hydraulic machines connected to each other with a stiff shaft, as shown in figure 10. While either one of them is driven as a pump, the second one is working as a motor. The shaft is modelled as a very stiff spring. This configuration is commonly used to separate hydraulic systems from each other for safety reasons, or as hydraulic transformers for energy recuperation. Simulating this model is problematic, because the shaft between the machines is very stiff compared to the two hydraulic circuits. This stiff connection limits the time step size for the whole model. A solution is to replace the two machines and the shaft with Modelica components.

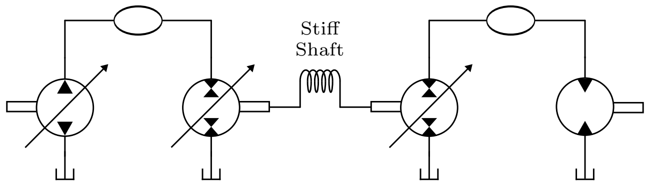

**Figure 10: Simulating two hydraulic circuits connected through a stiff shaft is problematic, because the shaft is much stiffer than the rest of the model.**

Quantifying the benefits from the solution is difficult. The results are strongly dependent on parameterisation, stiffness in the hydraulic systems and not least which tolerance for the stiffness that is considered acceptable. It is, however, clear from the experiments that it is not possible to simulate a rigid connection as a TLM element without introducing some sort of damping factor. With Modelica, however, a rigid shaft can be simulated without limiting the time step for the rest of the system. An alternative solution could be to model the two machines and the shaft as one single pre-compiled C++ component. This, however, would be far more complicated and offer limited reusability. It is also preferable that the user should be able to build the model from as small elements as possible.

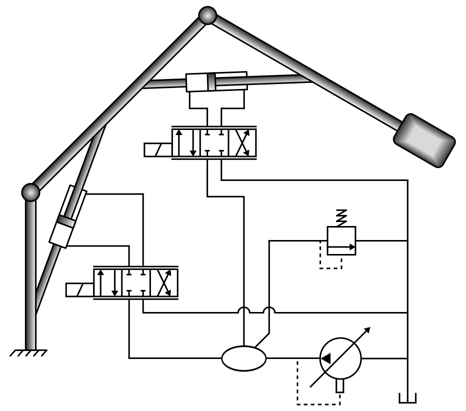

**Figure 11: A 2D-model of crane was modelled in Modelica, and connected to an actuation system modelled with TLM.**

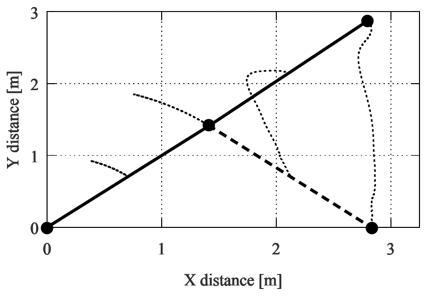

**Figure 12: Initial position (dashed), final position (solid) and movement paths for joints and centres (dotted) of the boom and jib in the crane model shown in figure 11.**

Finally, the third method mentioned in section 4 is verified using the model shown in figure 11. A two-dimensional crane is modelled using Modelica. It is controlled by a hydraulic servo system modelled using TLM. The boom and jib functions of the crane are controlled by one actuator each, connected according to the figure. The advantage of this is that a very stiff mechanical system and a significantly weaker system, such as a hydraulic circuit, can be simulated in the same model with decent performance.

Results are shown in figure 12. The load is lifted vertically from 0.0 m to 2.7 m. A fair comparison with a pure TLM system is difficult. Modelling 2D mechanics using TLM elements representing stiff joints was demonstrated by Krus [28]. Completely rigid joints are, however, not possible using TLM. Another approach could have been to model the mechanical system as one pre-compiled TLM component written in C++. It would on the other hand have been far more complicated than the Modelica approach, and the mathematical equations to solve would have been the same. The Modelica standard library also has models for multi-body mechanics, which would offer great reusability.

## 6. ANALYSIS AND DISCUSSION

The Modelica language has been implemented in a distributed-solver simulation tool. TLM elements are used to decouple equation systems. The user can explicitly place TLM elements wherever it is found to be physically motivated. Independent groups of components are then compiled separately. This method differs from automatic approaches in that the user retains the control over the model. It can also provide a better understanding of the underlying physical properties of the model.

An important benefit of the suggested method is the possibility of combine TLM with equation-based methods. This makes it easier for users who prefer TLM to cooperate with users who favour Modelica. The physically motivated decoupling is also often suitable for cooperation between different disciplines or departments within an organization. A significant future improvement would be to investigate compatibility with the Modelica Standard Library.

Simulating models using distributed solvers has several advantages over centralized solver methods. Each part of the model solving itself is a very intuitive approach. The solvers get an object-oriented structure, mirroring the model hierarchy. Another benefit is that it facilitates the use of multi-core architectures, since several sequential solvers can be used in parallel. Solving several smaller equation systems also requires less computation time than solving one large system in most cases.

Furthermore, the independency between sub-models makes it possible to test each one of them separately. A sub-model can be validated in a delimited context before being implemented in a larger model. This makes simulation results predictable and offers good robustness. It is easy to reproduce and locate errors and bugs. For example, the check that the number of variables equals the number of equations is performed for each sub-model. An error can thus be delimited to a smaller part of the model.

Event handling has not been considered in the current implementation. However, there are no technical constraints to implement this in the future. In addition, events need only be considered locally and do not propagate beyond the partition set up by the TLM elements. Global events would be more difficult, due to the distributed solver structure. While this may seem like a drawback, it should be pointed out that events immediately propagating to all parts of the system would be physically incorrect. In reality, information can never propagate infinitely fast, but is limited by the wave propagation speed.

A substantial advantage of distributed solver system simulation is the possibility to use different solver methods and step sizes in each individual sub-model. Stiff parts of the model can use more advanced solvers, lower tolerances and smaller step sizes, while less demanding parts can use simpler methods and larger time steps, see figure 13. How to choose an optimal distribution of time steps and solvers remains to be investigated. An even more complex approach would be to also let each individual sub-model use variable time steps, or even variable solver algorithms, during execution. These topics constitute an interesting future research challenge.

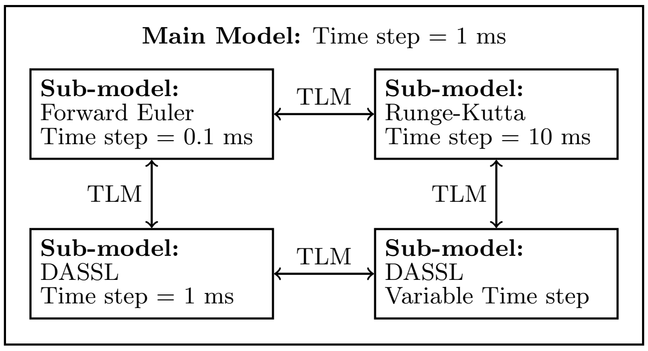
**Main Model:** Time step = 1 ms
**Sub-model:** Forward Euler Time step = 0.1 ms
**Sub-model:** Runge-Kutta Time step = 10 ms
**Sub-model:** DASSL Time step = 1 ms
**Sub-model:** DASSL Variable Time step
TLM
TLM
TLM
TLM

**Figure 13: An advantage with distributed solvers is the possibility to use different solver methods and step sizes in each sub-model.**

## 7. ACKNOWLEDGEMENTS

This work was supported by ProViking research school and the Swedish Foundation for Strategic Research (SSF).

## 8. REFERENCES

[1] Robert Braun and Petter Krus. Towards a parallel distributed equation-based simulation environment. In *53rd SIMS Conference on Simulation and Modelling*, Reykjavik, Iceland, 2012.

[2] Petter Krus. An automated approach for creating component and subsystem models for simulation of distributed systems. In *Proceeedings of the Nineth Bath International Fluid Power Workshop*, Bath, England, 1996.

[3] B. Johansson and P. Krus. Modelica in a Distributed Environment Using Transmission Line Modelling. In *Modelica 2000 Workshop*, Lund, Sweden, October 2000.

[4] Torsten Blochwitz, M Otter, M Arnold, C Bausch, C Clauß, H Elmqvist, A Junghanns, J Mauss, M Monteiro, T Neidhold, et al. The functional mockup interface for tool independent exchange of simulation models. In *Modelica'2011 Conference, March*, pages 20–22, 2011.

[5] Robert Braun and Petter Krus. Tool-independent distributed simulations using transmission line elements and the functional mock-up interface. In *54th SIMS Conference on Simulation and Modelling*, 2013.

[6] Kaj Nyström and Peter Fritzson. Parallel Simulation with Transmission Lines in Modelica. In *5th International Modelica Conference*, Vienna, Austria, September 2006.

[7] Francesco Casella. A strategy for parallel simulation of declarative object-oriented models of generalized physical networks. In *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools.* Linköping University Electronic Press, April 2013.

[8] Martin Sjölund, Mahder Gebremedhin, and Peter Fritzson. Parallelizing equation-based models for simulation on multi-core platforms by utilizing model structure. In *Proceedings of the 17th Workshop on Compilers for Parallel Computing*, July 2013.

[9] Alessandro Vittorio Papadopoulos and Alberto Leva.

Automating dynamic decoupling in object-oriented modelling and simulation tools. In *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press, April 2013.

[10] Abir Ben Khaled, Mongi Ben Gaid, and Daniel Simon. Parallelization approaches for the time-efficient simulation of hybrid dynamical systems: Application to combustion modeling. In *Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*. Linköping University Electronic Press, April 2013.

[11] Modelica and the Modelica Association. https://www.modelica.org, June 2014.

[12] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Computer Society Pr, 2006.

[13] D. de Cogan, W.J. O'Connor, and S. Pulko. *Transmission Line Matrix (TLM) in Computational Mechanics*. Taylor & Francis, 2005.

[14] D.M. Auslander. Distributed system simulation with bilateral delay-line models. *Journal of Basic Engineering*, pages 195–200, June 1968.

[15] P.B. Johns and M.A. O'Brian. Use of the transmission line modelling (T.L.M) method to solve nonlinear lumped networks. *The Radio And Electronic Engineer*, 50(1/2):59–70, 1980.

[16] Air Force Aero Propulsion Laboratory. Aircraft hydraulic system dynamic analysis. Technical report, Air Force Aero Propulsion Laboratory, 1977.

[17] A. Jansson and P. Krus. Real-time simulation using parallel processing. In *The Second Tampere International Conference On Fluid Power*, Tampere, Finland, 1991.

[18] J.D. Burton, K.A. Edge, and C.R. Burrows. Partitioned simulation of hydraulic systems using transmission-line modelling. *ASME WAM*, 1993.

[19] H. Le-Huy and J-C. Soumagne. Digital real-time simulation of transmission lines using parallel processors. *Mathematics and Computers in Simulation*, 38:293–301, 1995.

[20] K.K. Fung and S.Y.R. Hui. Transputer simulation of decoupled electrical circuits. *Mathematics and Computers in Simulation*, 42:1–13, 1997.

[21] R. Braun, P. Nordin, B. Eriksson, and P. Krus. High Performance System Simulation Using Multiple Processor Cores. In *The Twelfth Scandinavian International Conference On Fluid Power*, Tampere, Finland, May 2011.

[22] Alexander Siemers, Dag Fritzson, and Iakov Nakhimovski. General meta-model based co-simulations applied to mechanical systems. *Simulation Modelling Practice and Theory*, 17(4):612–624, 2009.

[23] P. Krus. Robust System Modelling Using Bi-lateral Delay Lines. In *Proceedings of the 2nd Conference on Modeling and Simulation for Safety and Security*, Linköping, Sweden, 2005.

[24] Robert Braun and Petter Krus. Multi-threaded real-time simulations of fluid power systems using transmission line elements. In *8th International Fluid Power Conference*, Dresden, Germany, March 2012.

[25] B. Eriksson, P. Nordin, and P. Krus. Hopsan NG, A C++ Implementation Using The TLM Simulation Technique. In *The 51st Conference On Simulation And Modelling*, Oulu, Finland, 2010.

[26] M. Axin, R. Braun, A. Dell'Amico, B. Eriksson, P. Nordin, K. Pettersson, I. Staack, and P. Krus. Next Generation Simulation Software Using Transmission Line Elements. In *Fluid Power and Motion Control*, Bath, England, October 2010.

[27] SymPy. http://sympy.org/, February 2012.

[28] Petter Krus. Modeling of mechanical systems using rigid bodies and transmission line joints. *Journal of dynamic systems, measurement, and control*, 121(4):606–611, 1999.

## APPENDIX

## A. MODELICA CODE FOR CAR MODEL

```
model CarsModel
    CarEngine engine(v_ref=10,Kp=1000,T_max=10000);
    CarWheel wheel1(Kr=10);
    CarWheel wheel2(Kr=10);
    CarWheel wheel3(Kr=10);
    CarWheel wheel4(Kr=10);
    CarWheel wheel5(Kr=10);
    CarWheel wheel6(Kr=10);
    CarWheel wheel7(Kr=10);
    CarWheel wheel8(Kr=10);
    CarChassi chassi1(M=1000,B=0,Kd=1);
    CarChassi chassi2(M=1000,B=0,Kd=1);

    TLM_Spring shaft(k=1000);
    TLM_Force force(F=0);
    TLM_Pulse pulse1(y_0=0,y_A=2000,t_0=35,t_1=35.5);
    TLM_Pulse pulse2(y_0=0,y_A=2000,t_0=35,t_1=35.5);
    TLM_Pulse pulse3(y_0=0,y_A=2000,t_0=36,t_1=36.5);
    TLM_Pulse pulse4(y_0=0,y_A=2000,t_0=36,t_1=36.5);
    TLM_Pulse pulse5(y_0=0,y_A=2000,t_0=40,t_1=40.5);
    TLM_Pulse pulse6(y_0=0,y_A=2000,t_0=40,t_1=40.5);
    TLM_Pulse pulse7(y_0=0,y_A=2000,t_0=41,t_1=41.5);
    TLM_Pulse pulse8(y_0=0,y_A=2000,t_0=41,t_1=41.5);

equation
    connect(wheel1.P1,chassi1.P1);
    connect(wheel2.P1,chassi1.P2);
    connect(wheel3.P1,chassi1.P3);
    connect(wheel4.P1,chassi1.P4);

    connect(wheel5.P1,chassi2.P1);
    connect(wheel6.P1,chassi2.P2);
    connect(wheel7.P1,chassi2.P3);
    connect(wheel8.P1,chassi2.P4);

    connect(chassi2.PD,force.P1);
    connect(chassi1.PM,engine.P1);

    connect(chassi2.PM,shaft.P1);
    connect(shaft.P2,chassi1.PD);

    connect(pulse1.out,wheel1.P2);
    connect(pulse2.out,wheel2.P2);
    connect(pulse3.out,wheel3.P2);
    connect(pulse4.out,wheel4.P2);

    connect(pulse5.out,wheel5.P2);
    connect(pulse6.out,wheel6.P2);
    connect(pulse7.out,wheel7.P2);
    connect(pulse8.out,wheel8.P2);
end CarsModel;
```

**Listing 2: Modelica code for the first example model.**

## B. MODELICA CODE FOR LEFT CAR

```
model ModelicaTemp879987
    annotation(hopsanCqsType = "Q");

    NodeMechanic Force_P1;
    NodeSignalIn Pulse_1_out;
    NodeSignalIn Pulse_3_out;
    NodeMechanic Rope_P1;

    [...]

equation
    chassi_Fd=chassi_Kd*pow(chassi_PM_v,2.0);
    der(chassi_PM_v)*chassi_M+chassi_PM_v*chassi_B=
         chassi_PM_f-chassi_P1_f-chassi_P2_f-
         chassi_P3_f-chassi_P4_f-chassi_PD_f-chassi_Fd
         ;
    chassi_P1_v=chassi_PM_v;
    chassi_P2_v=chassi_PM_v;
    chassi_P3_v=chassi_PM_v;
    chassi_P4_v=chassi_PM_v;
    chassi_PD_v=-chassi_PM_v;
    chassi_PM_v=der(chassi_PM_x);
    chassi_P1_v=der(chassi_P1_x);
    chassi_P2_v=der(chassi_P2_x);
    chassi_P3_v=der(chassi_P3_x);
    chassi_P4_v=der(chassi_P4_x);
    chassi_PD_v=der(chassi_PD_x);

    wheel1_P1_f=-wheel1_P1_v*wheel1_Kr-wheel1_P2_y;
    wheel2_P1_f=-wheel2_P1_v*wheel2_Kr-wheel2_P2_y;
    wheel3_P1_f=-wheel3_P1_v*wheel3_Kr-wheel3_P2_y;
    wheel4_P1_f=-wheel4_P1_v*wheel4_Kr-wheel4_P2_y;

    wheel2_P1_v=chassi_P2_v;
    wheel2_P1_x=chassi_P2_x;
    wheel2_P1_f=-chassi_P2_f;

    wheel3_P1_v=chassi_P3_v;
    wheel3_P1_x=chassi_P3_x;
    wheel3_P1_f=-chassi_P3_f;

    wheel4_P1_v=chassi_P4_v;
    wheel4_P1_x=chassi_P4_x;
    wheel4_P1_f=-chassi_P4_f;

    chassi_PD_v=Force_P1.v;
    chassi_PD_x=Force_P1.x;
    chassi_PD_f=Force_P1.f;

    Pulse_1_out.y=wheel2_P2_y;
    Pulse_1_out.y=wheel4_P2_y;
    Pulse_3_out.y=wheel1_P2_y;
    Pulse_3_out.y=wheel3_P2_y;

    wheel1_P1_v=chassi_P1_v;
    wheel1_P1_x=chassi_P1_x;
    wheel1_P1_f=-chassi_P1_f;

    chassi_PM_v=Rope_P1.v;
    chassi_PM_x=Rope_P1.x;
    chassi_PM_f=Rope_P1.f;
```

TLM equations

```
    Force_P1.f=Force_P1.c+Force_P1.v*Force_P1.Zc;
    Rope_P1.f=Rope_P1.c+Rope_P1.v*Rope_P1.Zc;
end ModelicaTemp879987;
```

**Listing 3: Modelica code for one of the cars in the first example model. The automatically added TLM equations are marked with a box.**