Supporting Dimensional Analysis in SystemC-AMS

Torsten Maehne
torsten.maehne@epfl.ch

Alain Vachoux
alain.vachoux@epfl.ch

Laboratoire de Systèmes Microélectroniques (LSM)
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
Phone: +41(21)69-36922, Fax: +41(21)69-36959, WWW: http://lsm.epfl.ch/

ABSTRACT

This paper will introduce new modeling capabilities for SystemC-AMS to describe energy conserving multi-domain systems in a formal and consistent way at a high level of abstraction. To this end, all variables and parameters of the system model need to be annotated with their measurement units in such a way that they become intrinsic part of the data type. This enforces correct model assembly through strict interfaces and coherent formulas describing the analog behavior by means of dimensional analysis. A library of generic block diagram components has been developed to demonstrate how both requirements can be met using the Boost libraries together with SystemC-AMS. The demonstrated implementation techniques are the key to integrate new Models of Computation (MoCs) into SystemC-AMS to facilitate further the description of multi-domain systems.

1. INTRODUCTION

Today's Systems-on-Chips (SoCs) become more and more heterogeneous to interact closer with their analog environment. To achieve this, Analog and Mixed-Signal (AMS) blocks including MEMS actuators and sensors, power electronics, and RF interfaces are added to and functionally interwoven with the digital hardware and software. Their design requires the cooperation of different domain experts, who employ different modeling formalisms and tools. To understand the functional interaction of the different system parts involving potentially different physical domains and thus to be able to refine the system architecture and to derive consistent component specifications, a *system model* needs to be created early on in the design process and continuously refined from functional to architectural abstraction levels. Thus, an *executable specification* is created, which later helps to verify the correct integration of the individually designed components to a system.

A successful integration of heterogeneous parts requires a strict specification of the interfaces between the system components, which should be expressible in the system model. This includes not only the *data type* (storage format) of the *value* of a *quantity* exchanged through ports or used to parametrize the component, but also the semantical information on how the value is to be interpreted. The latter is expressed by the *measurement unit*, which itself expresses the abstract *dimension* of measure and the actual *system of units* used to measure the amount of the quantity. Their usage is common practice in any natural science and engineering field for hand calculations and written documentation. It permits to check the coherency of the equations used to describe the system by means of *dimensional analysis*. However, most common modeling and programming languages permit only to specify the data type of the value and not of the whole quantity during the declaration of constants, variables, ports, and parameters. This is less a problem in single domain systems designed primarily by one engineering discipline, as the correct interpretation of the data as a quantity can be supported by using a consistent naming scheme based on established symbols or acronyms and using always the same measurement unit for a dimension and annotating its usage in the comments. These measures lower the risk of implementing incoherent equations describing the system behavior, which lack, e.g., a conversion factor or accidentally sum up quantities of different dimensions, and interconnecting wrongly the system components.

However, in multi-domain systems the risk for misinterpreting a quantity is rising due to partially conflicting standards for quantity symbols (e.g., v for voltage and speed) in different engineering disciplines and different common practices regarding the units (e.g., feet and meter for a distance) and scale factors (e.g., μm) to measure them. This can lead to very hard to spot problems, which may stay undiscovered as the simulation results seem to be meaningful and in the expected order of magnitude. This is especially true when integrating Intellectual Properties (IPs) from different sources and reusing legacy models with potentially diverting specifications. For example, one of the root causes for the loss of NASA's Mars Climate Orbiter in 1999 during the orbit entering maneuver was a forgotten unit conversion between imperial and *Système International d'unités* (SI) units between different programs used for the course corrections [13]. However, the parallel usage of different systems of units in a single model might be necessary. This can be illustrated by the VHDL-AMS model of a micromechanical yaw rate sensor system presented in [12].

The reduced-order model of the sensor was extracted from an FEM model of the micromechanical structure, which used for numerical reasons the μMKSV system of units (displacements in μm, forces in μN, etc.). The electrical part of the system model responsible for the control of the mechanical structure's movements used the SI system of units, which required the insertion of unit converters at the subsystem boundaries.

AMS-hardware description languages such as VHDL-AMS or Verilog-AMS have limitations regarding the support of dimensional analysis. VHDL-AMS only offers a way to annotate quantities with their units for presentation purposes. The full support of dimensional analysis was considered during language design, but rejected as the required changes to the type system would have rendered it incompatible with VHDL [7]. Other more recent efforts to support quantity types and dimensional analysis include Modelica [2] and the programming language F# [10]. Both approaches make units part of the language syntax and implement dedicated support for dimensional analysis in the compilers and other development tools.

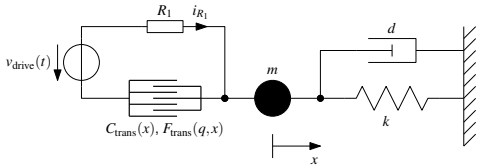
The mentioned languages have their limitations in the modeling of the software/hardware interaction, in the support of dedicated MoCs for heterogeneous system modeling, and the reuse of legacy code and models, which require the usage of a very flexible and extensible simulation framework. This is the strength of the C++-based simulation framework *SystemC* [6]. It supports the description of digital hardware/software systems from functional down to register transfer level by using the *Discrete Event* (DE) *Model of Computation* (MoC). The openness of this environment facilitates the integration of other libraries and legacy code and allows the implementation of new modeling formalisms. For example, the Open SystemC Initiative (OSCI) AMS Working Group (AMSWG) is on the way to standardize *AMS extensions to SystemC* [5, 14]. The added *Timed Data Flow* (TDF), *Linear Signal Flow* (LSF), and *Electrical Linear Network* (ELN) MoCs allow the description of analog behaviors at different levels of abstraction. This work uses a prototype implementation called *SystemC-AMS* [16], which implements a precursor of the TDF MoC called *Synchronous Data Flow* (SDF).

The work presented in this paper aims to improve the modeling capabilities of SystemC-AMS for energy conserving multi-domain components to allow their description in a formal and consistent way by means of dimensional analysis. Section 2 discusses how a multi-domain system can be modeled on successively higher abstraction levels. It shows how the loss of the semantic information due to the usage of more abstract and generic modeling primitives can be compensated by conserving the link to the physical domain by annotating the model's signals and parameters with their measurement units. Section 3 describes how quantity types and dimensional analysis can be integrated into SystemC-AMS to achieve the stated goal. As an application example, an electromechanical transducer driving a micromechanical resonator is modeled using this library in Section 4.

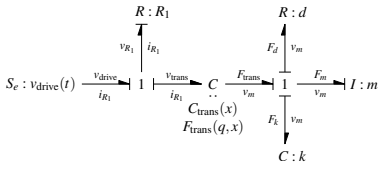
2. MULTI-DOMAIN SYSTEM MODELING

In this section, we will show how to derive a SystemC-AMS compatible model of a multi-domain system by successively rising the abstraction level without loosing the link to the physical domain. We will use an electromechanical transducer (electrostatic comb-drive actuator) linked to a micromechanical resonator as an example (Figure 1). In a first step, the system can be modeled using *domain-specific primitives*, in our case (Figure 1a): voltage source, resistor for the electrical domain and mass, spring, damper for the mechanical domain. The comb drive actuator is part of both domains and acts electrically as a capacitor, which capacitance $C_{\text{trans}}(x)$ depends on the current displacement x, and mechanically as a force source $F_{\text{trans}}(q, x)$, which value depends on the electrical charge q stored on the capacitor and the displacement x. The resulting model accurately represents the physical structure of the system. However, this approach requires a simulator to provide model implementations for each primitive of each supported physical domain. This implies an overhead as there exist analogies between the primitives of different domains (e.g., resistor/damper, capacitor/spring, inductor/mass). This has been exploited regularly to, e.g., simulate mechanical resonators using an electrical simulator such as SPICE. However, this approach sacrifices clarity, which easily leads to modeling mistakes. SystemC-AMS currently only implements linear electrical primitives on this abstraction level.

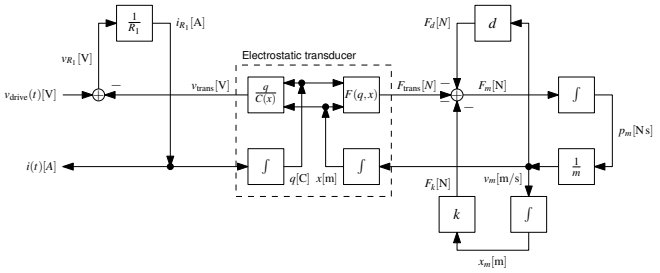
The *bond graph formalism* [9] is taking advantage of these analogies to unify the description of multi-domain systems through a reduced set of *generic primitives*. The domain-specific description (e.g., electrical circuit, mechanical multi-body system, etc.) is mapped in a systematic way on a graph (Figure 1b) describing the energy flow between primitives modeling energy sources (S_e, S_f), resistive/capacitive/inertial behavior (R, C, I), quantity transformations (TF, GY), and energy distribution through junctions (0, 1). The energy link between the ports of two primitives Pr_i and Pr_k is represented with an half-arrow shaped *bond*: $Pr_i \xrightarrow[f]{e} Pr_k$. Associated to each bond are an *effort* e and a *flow* f *variable*. They are called *power variables* because their product is the power P and are domain-specific, e.g., voltage v and current i for the electrical domain and force F and velocity v for the translational domain, respectively. Thus, the link to the physical domain is not anymore kept through the name of the primitive, but through the quantity type (measurement unit) of each variable and parameter. The *power flow direction* for positive e and f is indicated by the half-arrow. The describing equations of the primitives and their interconnection rules through common-effort (0) or common-flow (1) junctions guarantee the conservation of energy and thus the description of a phys-



(a) Domain-specific model.



(b) Equivalent bond graph model.



(c) Derived block diagram model.

Figure 1: Equivalent models of an electromechanical transducer linked to a micromechanical resonator.

ical system. SystemC-AMS does not yet support the bond graph formalism, but there is an effort in this direction [11].

The energy exchange through the bond between two primitives causes the effort and flow variables to act in opposite directions. This can be used to determine the computational direction and is indicated by a perpendicular *causal stroke* at one end of the bond. It states that at this side the effort variable e is known (it acts as an input) and thus $f := \Phi_k^{-1}(e)$. Consequently, the flow f is known on the other side of the bond thus that the effort: $e := \Phi_i(f)$. The equations describing the component behavior impose a *required*, *preferred* (e.g., due to numerical reasons), or *free* causality (*effort-in* or *effort-out*) on the element ports. The resulting constraints need to be propagated to all related ports, e.g., using the *Sequential Causality Assignment Procedure* [9].

The causality assignment allows also for a natural integration of bond graphs with block diagrams and their transformation in the latter (Figure 1c). The *block diagram formalism* does not guarantee itself the conservation of energy in the system. It is emulated by the way the block diagram primitives are interconnected. The block diagram primitives (scaler, integrator, summer, etc.) are so generic that they don't establish a direct link to the modeled physical effect. That's why it becomes paramount to annotate each signal and parameter with its quantity type to reestablish the link and to be able to check the system model on the structural and equation level for consistency. Block diagram models of physical components are in general hard to reuse as all physical quantities have already been assigned their input/output roles till the interface of the component. This limits seriously the interconnection options with other physical components (e.g., only series connection or only parallel connection). However, they are entirely causal, which allows for a procedural and thus very efficient model execution, e.g., with the SDF MoC of SystemC-AMS [16].

3. INTEGRATING DIMENSIONAL ANALYSIS INTO SYSTEMC-AMS

The integration of dimensional analysis into SystemC-AMS requires two steps: first, to implement a quantity type with unit annotation and dimensional analysis in C++ without prohibitive runtime penalty; and second, to facilitate the consistent usage of this quantity type in SystemC-AMS models of a multi-domain system by offering a library that helps to reduce code overhead due to the reimplementation of very similar behaviors for different quantity types.

For the first step, a mature implementation is available in form of the peer-reviewed *Boost.Units* library [15], which has been used in this work. Using the template metaprogramming technique [1], it implements dimensional analysis for arbitrary systems of units at compile-time as part of the static type checking phase without requiring modifications to the compiler or an additional tool. The library represents an arbitrary *composite unit* with the help of the class unit<Dim, System>. Its two template arguments encode in static type lists the *dimension* as a reduced ordered *set of base dimensions raised to a rational power* and the associated *system of units* that defines the set of base dimensions and their measure. For example, the energy is represented through $[\mathrm{M}]^1[\mathrm{L}]^2[\mathrm{T}]^{-2}$ using the base dimensions mass [M], length [L], and time [T], which is in SI: $\mathrm{kg\,m^2\,s^{-2}} = \mathrm{N\,m} = \mathrm{J}$. The compile-time derivation of new units due to arithmetical operations with units, i.e., the dimensional analysis, is done through traits classes. The unit type U and value type V (by default **double**, but maybe, e.g., std::complex<T>) form a unique type quantity<U, V>, which overloads only the legal arithmetic and assignment operators. Thus, the compiler issues a "missing overload" error for illegal operations, e.g., the sum of two quantities with different units or the assignment between incompatible quantities. Numerical constants are annotated in the source code by multiplying them with their measurement unit, e.g.:

```
quantity<si::energy> E = 1.5 * si::newton * si::meter;
```

where si::newton and si::meter are static constants of type unit<force_dimension, si::system> and unit<length_dimension, si::system>, respectively. Their multiplication with a **double** value yields the correct type quantity<si::energy, **double**> for the assignment to the variable E. The example shows that the complex infrastructure of types encoding all properties of a quantity (dimension, system of unit, value type) is mostly

hidden from the user. However, compiler errors can get very cryptic as they usually state the fully expanded type name. Boost.Units also overloads all standard mathematical functions defined in <cmath> for the new quantity type taking into account any unit transformations by the functions. As the unit is encoded into the quantity only as part of the type and not as a member variable, modern C++ compiler can optimize this information away after the type checking phase leaving behind an object with the same memory layout as the value type. Thus, no runtime penalty is caused by doing arithmetics with quantities. Only the compile time is increased.

For the second step, we use the existing DE and SDF MoCs of SystemC-AMS [16], as they allow to parametrize the data types of the involved ports and signals. The SDF MoC is suitable to simulate block diagram models of multi-domain systems as derived in Section 2. It processes a synchronous data flow of time-annotated samples with the help of SDF modules, which read at each simulation time step a predefined number of samples from their input ports, process them, and write the results as a series of samples to the output ports. The SDF MoC statically schedules the execution order of the interconnected SDF modules. Converter ports interface/synchronize with the surrounding DE modules.

To demonstrate the feasibility and power of the combination of both approaches, a library of basic block diagram components has been implemented called scax_block_diagram (Table 1). As the function carried out by each block diagram component is orthogonal to the actual type of the inputs and outputs, each SDF module is implemented as a template class, which allows to parametrize its port and parameter types upon instantiation. Dependencies between these types due to the implemented model equations are either automatically satisfied or at least checked for consistency using traits classes or static assertions, respectively. The new models are not constrained to work exclusively with quantity<U, V> types, but can also be parametrized for other types such as **double**. This ensures interoperability with other user-created SDF modules. To render the library even more flexible, some modules, such as the SDF source and the different function modules, take a function as argument upon instantiation. Thus, e.g., the waveform to generate or the function to apply to the read input signals to transform them to an output sample can be specified.

Table 1: Block diagram modules provided by scax_block_diagram**.**

Name	Description
scax_source<T,TimeType>	SDF samples source module using a waveform function: f : TimeType → T
scax_sink<T>	SDF samples sink module
scax_scale<T1,T2>	Scale module
scax_sum<T>	Summing module with variable input number
scax_mul<T1,T2>	Multiplier module with two inputs
scax_func1<T1,T2>	Time-independent function module with one input: f : T1 → T2
scax_func2<T1,T2,T3>	Time-independent function module with two inputs: f : T1 × T2 → T3
scax_func3<T1,T2,T3,T4>	Time-independent function module with three inputs: f : T1 × T2 × T3 → T4
scax_func1t<T1,T2,TimeType>	Time-dependent function module with one input: f : T1 × TimeType → T2
scax_func2t<T1,T2,T3,TimeType>	Time-dependent function module with two inputs: f : T1 × T2 × TimeType → T3
scax_integ_trapez<T>	Trapezoidal integrator module
scax_dot_secant<T>	Differentiator module using asymmetric evaluation of Newton's difference quotient

4. APPLICATION EXAMPLE

As an application example, the block diagram model of the electromechanical transducer linked to a micromechanical resonator (Section 2, Figure 1c) has been implemented using the developed block diagram library.

The transducer is encapsulated in an own sc_module elmech_transducer (Listing 1), which can be parametrized upon instantiation with the voltage and force transfer functions (v_func corresponding to $\frac{q}{C(x)}$ and F_func corresponding to $F(q,x)$ in Figure 1c, respectively) by passing them along with the initial conditions as constructor arguments. Compared to the purely computational data type **double**, the usage of quantity types strengthens the signature of the argument list so that the compiler can detect, if parameters are accidentally passed in the wrong order, e.g., for the initial charge q_0 and displacement x_0 of the transducer. As the transducer model is entirely structural, it contains only block diagram component instances and specifies their interconnection by binding their ports to signals in the constructor. Typedefs are used to facilitate the usage of the different quantity types (in our case, e.g., voltage_t, force_t). The specification of the quantity type for the ports v_out and v_in make them recognizable as a voltage output and speed input, respectively, despite the overlapping symbols and without the need of comments. The compiler detects any binding of incompatible quantity signals to ports. This would not be the case, if only the computational data type **double** would have been used for the signals and ports.

Listing 1: Electromechanical transducer module.

```
using namespace std; using namespace std::tr1;
using namespace boost::units; namspace si = boost::units::si;
using namespace scax_bd;

// Block diagram model of an electromechanical transducer
class elmech_transducer : public sc_core::sc_module {
public:
```

```
  // Typedefs for common quantity types
  typedef quantity<si::electric_potential> voltage_t;
  typedef quantity<si::force> force_t;
  // ...
  // Electrical and mechanical ports
  sca_sdf_in<current_t> i_in;
  sca_sdf_out<voltage_t> v_out;
  sca_sdf_in<velocity_t> v_in;
  sca_sdf_out<force_t> F_out;

  // Construct structural transducer model and set the force
  // and voltage functions as well as the initial conditions
  elmech_transducer(
    const sc_core::sc_module_name& name,
    function<voltage_t (charge_t, displacement_t)> v_func,
    function<force_t (charge_t, displacement_t)> F_func,
    charge_t q_0 = 0.0 * si::coulomb,
    displacement_t x_0 = 0.0 * si::meter)
    : i_in("i_in"), v_out("v_out"), v_in("v_in"), F_out("F_out"),
      i_integ(new scax_integ_trapez<current_t>("i_integ", q_0)),
      v_integ(new scax_integ_trapez<velocity_t>("v_integ", x_0)),
      v_func2(new scax_func2<charge_t, displacement_t,
                             voltage_t>("v_func2", v_func)),
      F_func2(new scax_func2<charge_t, displacement_t,
                             force_t>("F_func2", F_func))
  {
    // Specify connectivity
    i_integ->in(i_in); i_integ->out(q_sig);

    v_integ->in(v_in); v_integ->out(x_sig);

    v_func2->in1(q_sig); v_func2->in2(x_sig);
    v_func2->out(v_out);

    F_func2->in1(q_sig); F_func2->in2(x_sig);
    F_func2->out(F_out);
  }

private:
  // Internal signals
  sca_sdf_signal<charge_t> q_sig;
  sca_sdf_signal<displacement_t> x_sig;

  // Internal SDF modules from block diagram library
  auto_ptr<scax_integ_trapez<current_t> > i_integ;
  auto_ptr<scax_integ_trapez<velocity_t> > v_integ;
  auto_ptr<scax_func2<charge_t, displacement_t, voltage_t> > v_func2;
  auto_ptr<scax_func2<charge_t, displacement_t, force_t> > F_func2;
};
```

Listing 2 shows code extracts from the test bench for the transducer and resonator. Being also a structural model, it follows the same approach as the elmech_transducer model. What is new is the definition of quantity constants, which are used as instance parameters (e.g., R_1, k). The usage of quantity types prevents any accidental assignment of a quantity constant of the right dimension but wrong system of unit without proper conversion, e.g., the usage of pound instead of kilogram for the mass m. The transfer functions (v_trans_func, F_trans_func) of the transducer module instance are realized as function objects (functors), which interface (types of arguments and return value) is defined with the help of the *Boost.Function* library [4]. The functions themselves are defined in-place with the placeholders _1 and _2 representing the two arguments. This compact notation is made possible by the *Boost.Lambda* library [8]. The interface code between it and Boost.Units was developed as part of the presented project. Due to the usage of quantity types in the function interface, a contract is formed, which fulfillment by the user-supplied implementation can be checked by the compiler. Mixing up the order of the quantity arguments or forgetting a term in the implemented functions is detected in most cases due to the dimensional analysis, as it will usually either lead to a conflict on the level of mathematical operations or result type due to incompatible units. The waveform of the driving voltage source v_drive_src is specified slightly differently by passing a functor instance of type scax_pulse<T> parametrized with the return type voltage_type and the parameters of the pulse waveform. scax_pulse<T> is part of a generic waveform generator library also developed for the project.

Listing 2: Test bench for the transducer and resonator.

```
#include "systemc-ams.h"
// ...
int sc_main(int argc, char* argv[]) {
  // ...
  // Derivation of new units
  typedef divide_typeof_helper<si::force, si::length>::type stiffness;
  typedef divide_typeof_helper<si::force, si::velocity>::type
    viscous_damping;

  // Electrical component constants
  const quantity<si::resistance> R_1 = 50.0e3 * si::ohm;
  const quantity<si::capacitance> C_trans_0 = 500.0e-12 * si::farad;
  const quantity<si::length> overlap = 20.0e-6 * si::meter;
  // Mechanical component constants
  const quantity<si::mass> m = 10.0e-9 * si::kilogram;
  const quantity<stiffness> k = 100.0 * si::newton / si::meter;
  const quantity<viscous_damping>
    d = 50.0e-6 * si::newton * si::second / si::meter;

  // Transducer capacitance formula v_trans(q, x)
  function<voltage_type (charge_type, displacement_type)>
    v_trans_func = _1 / (C_trans_0 * (1.0 - (_2 / overlap)));
  // Transducer force formula F_trans(q, x)
  function<force_type (charge_type, displacement_type)>
    F_trans_func = (_1 * _1 * overlap)
                   / (2.0 * C_trans_0 * (overlap-_2) * (overlap-_2));

  // Initial conditions and stimuli
  // ...
  // Signals
  sca_sdf_signal<voltage_type> v_drive("v_drive");
  // ...
  sca_sdf_signal<momentum_type> p_m("p_m");
  sca_sdf_signal<displacement_type> x_m("x_m");

  // Electrical driving circuit
  scax_source<voltage_type>
    v_drive_src("v_drive_src", scax_pulse<voltage_type>(
      V_drive_0, V_drive_1, t_drive_delay, t_drive_rise,
      t_drive_fall, t_drive_pulse, t_drive_period));
  v_drive_src.out(v_drive);
  // ...
  // Electromechanical transducer
  elmech_transducer transducer("transducer", v_trans_func, F_trans_func,
                               q_trans_0, x_m_0);
  transducer.i_in(i_R_1); transducer.v_out(v_trans);
  transducer.v_in(v_m); transducer.F_out(F_trans);
  // Mechanical resonator
```

```
  // ...
  // Tracing and simulation
  // ...
  v_drive_src.out.set_T(t_step);
  sc_start(t_sim);
  return 0;
}
```

Figure 2 shows the simulation results with SystemC-AMS. The pulsed input voltage v_{drive} excites the sinusoidal oscillation of the mechanical resonator. Frequencies other than its natural resonance frequency are filtered out due to its high quality factor. The common mode of the driving voltage shows up in the resulting electrostatic force and displacement of the resonator. A variant of the model using **double** instead of quantity<U, V> yields, as expected, numerically the same results, however its code required much more comments to compensate for the semantical loss of quantity types and units. Reference models for the example developed in VHDL-AMS showed the same dynamic behavior as the SystemC-AMS models. This demonstrates the successful simulation of all major effects of this small system. Table 2 compares the compile and simulation times for the quantity<U, V> and **double** model variants. It can be seen that the usage of Boost.Units significantly increases the compile time, but hardly affects the simulation time. This is the price to pay for the achieved stronger checking of the model interfaces and equations.

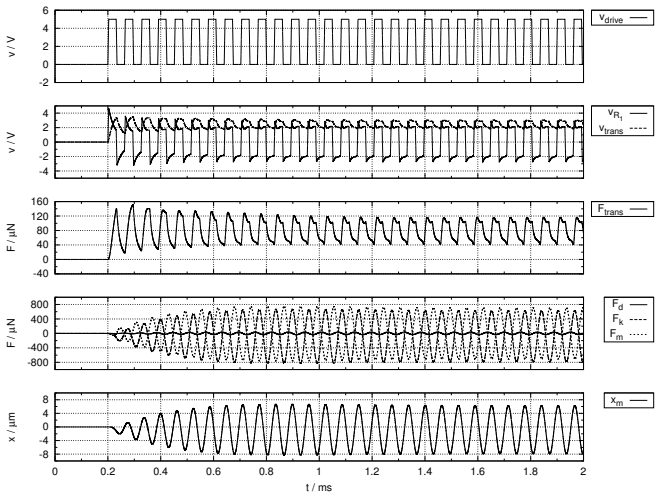


Figure 2: Simulation results of the block diagram in Figure 1c.

Table 2: Comparison of the compilation and simulation times for the electromechanical tranducer example implemented in SystemC-AMS once with Boost.Units quantity types and once with double. The simulated time was 2.4 ms **with a fixed time step of** 10 ns**. The measurements were done on an Intel Pentium 4** 3 GHz**,** 1 MB **Cache,** 2 GB **RAM, Linux 2.6.26 (i386).**

Language/Simulator	Feature	$t_{compile}$	$t_{simulate}$
SystemC-AMS 0.15 RC5	quantity<U, V>	19.07 s	11.76 s
(+ SystemC 2.2.0)	double	9.69 s	11.38 s

5. CONCLUSIONS AND OUTLOOK

This paper showed how relevant dimensional analysis is to ensure consistent model equations and proper assembly of models for multi-domain systems. It described how it can be implemented into C++ without modifying the language itself or imposing a runtime penalty. Its integration with SystemC-AMS was demonstrated by developing a flexible library of block diagram components for the SDF MoC enabling analog modeling at a high abstraction level without loosing the link to the physical domain. The need for such kind of library has been expressed in the requirements specification [3] of the OSCI AMS draft 1 standard [14]. In a next step, the implementation techniques tested during the development of this library will be applied to the bond graph MoC under development for SystemC-AMS [11]. Physical domains will be implemented using traits classes to specify the proper quantity types, which represent effort, flow, power, and energy in the electrical, mechanical, and other domains. The model equations of the bond graph primitives will automatically ensure that only consistent domains can be defined.

6. ACKNOWLEDGMENTS

This work has been funded by the Hasler-Stiftung under project № 2161.

7. REFERENCES

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming*. Addison-Wesley Professional, 2004.

[2] D. Broman, P. Aronsson, and P. Fritzson, "Design considerations for dimensional inference and unit consistency checking in Modelica," in *Proc. 6th Int. Modelica Conf.*, 2008.

[3] K. Einwich, *et al.*, "Requirements specification for SystemC analog mixed signal (AMS) extensions," OSCI, 2008.

[4] D. Gregor, *Boost.Function*, 2001-2004.

[5] C. Grimm, *et al.*, "An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions," OSCI AMSWG, 2008.

[6] *IEEE Standard 1666-2005, SystemC Language Reference Manual*, IEEE, Mar. 2006.

[7] IEEE 1076.1 Language Design Committee, "Meeting minutes of September 19–22, 1995," Brighton, UK.

[8] J. Järvi, *Boost.Lambda*, 1999-2004.

[9] D. C. Karnopp, D. L. Margolis, and R. C. Rosenberg, *System Dynamics: Modeling and Simulation of Mechatronic Systems*, 4th ed. Wiley, Jan. 2006.

[10] A. J. Kennedy, "Types for units-of-measure: Theory and practice," in *Proc. CEFP 2009*.

[11] T. Maehne, A. Vachoux, and Y. Leblebici, "Development of a bond graph based model of computation for SystemC-AMS," in *Proc. PRIME 2008*.

[12] T. Mähne, *et al.*, "Creating virtual prototypes of complex MEMS transducers using reduced-order modelling methods and VHDL-AMS," in *Applications of Specification and Design Languages for SoCs*. Springer, 2006.

[13] J. Oberg, "Why the Mars probe went off course," *IEEE Spectrum Magazine*, vol. 36, no. 12, Dec. 1999.

[14] *Draft Standard SystemC AMS Extensions Language Reference Manual*, OSCI, 2008.

[15] M. C. Schabel and S. Watanabe, *Boost.Units 1.0.0*, 2003-2008.

[16] A. Vachoux, C. Grimm, and K. Einwich, "Extending SystemC to support mixed discrete-continuous system modeling and simulation," in *Proc. ISCAS 2005*.