Automatic Dimensional Consistency Checking for Simulation Specifications

Mikael Sandberg
Mälardalen University
Västerås, Sweden
mikael.sandberg@mdh.se

Daniel Persson
Mälardalen University
Västerås, Sweden
dpn99004@idt.mdh.se

Björn Lisper
Mälardalen University
Västerås, Sweden
bjorn.lisper@mdh.se

ABSTRACT

Simulation specification languages usually have support for units or dimensions, but seldom use it for more then presenting simulation results. We will show that this annotation can be used to analyze the specifications and as a result eliminate dimensional errors in the specification equations and expressions.

We use well known theories on dimensions and type systems to achieve a sound and complete analysis method. We also extend known analysis methods and thereby the set of constructs that can be analyzed. The algorithms we have developed are general and can be implemented for a wide variety of simulation languages.

In this paper we present a prototype implementation of our dimensional consistency analysis method for a widely used simulation language, gPROMS. The prototype is able to analyze the dimensional consistency as well as to infer missing dimensions in simulation specifications for gPROMS. Furthermore, our tool can suggest to the simulationists, if needed, what parameters/variables should be annotated with dimensions to make all dimensions uniquely determined by the tool.

1. INTRODUCTION

We have developed a method to validate the dimensional correctness of simulation specifications in languages such as Modelica [6] and gPROMS [14]. Such languages specify models by differential algebraic equations in a structured or object-oriented paradigm.

Some simulation modelling languages have support for dimensional properties by enabling the simulationists to specify units for simulation output, but this information is seldom used for more than displaying units in graphs and tables. We are trying to show that a *Dimensional Inference* (*DI*) system will greatly enhance the usability of such languages and help produce models that are dimensionally consistent and thus more likely to be correct in general.

Dimensions (e.g length, mass and force) are used to validate equations in physics and other scientific domains. If an equation cannot be validated with respect to its dimensional correctness, it is most likely not correct in any aspect. Thus, validating dimensions can catch design errors early in the modelling process.

We exploit this analysis method, called *Dimensional Analysis* (*DA*), by automating it and combining it with type inference, see Section 1.2. The strength of the two combined methods make up an appealing analysis method that we like to call Dimensional Inference.

The automation of the analysis method enables us to validate whole specifications and guarantee their dimensional correctness before any simulation is performed. We believe that our analysis method greatly increases the quality of the models by ensuring dimensional consistency. The quality increase is gained early in the development process (e.g. before any simulation is performed). The analysis method is also fairly fast, consistent and sound.

Simulationists are faced with increasingly more complex models to simulate. A tool that increases the productivity as well as the quality of models is worth while. We propose that a dimensional correctness analysis method is incorporated into the development cycle of model development and simulation.

We have developed a stand-alone tool of our system that analyzes gPROMS simulation specifications. It will effectively determine if a gPROMS specification is dimensionally consistent or report what equations do not fulfill the dimensional correctness criteria. We use specially tagged comments in gPROMS to specify the dimensions of parameters and variables. Another tool that will analyze Modelica specifications is under development.

1.1 Dimensional theory

Dimensions can be seen as elements in an Abelian group, where the generating elements are the base dimensions. For instance, in mechanics we have three base dimensions: Mass (M), Length (L) and Time (T). Each dimension occurring in mechanics is then a "product" of these, like Force ($M^1L^1T^{-2}$). Equivalently, each dimension can be represented by a vector of base dimension exponents. For instance, the dimension of Force is represented by $\langle 1, 1, -2\rangle$. "Multiplication" in the Abelian group corresponds to addition of the corresponding dimension vectors. Consider, for instance, the famous Newton's second law:

$$F = ma \tag{1}$$

The entities in (1) have the following dimensions:

	F	m	a
Group	$M^1L^1T^{-2}$	M^1	L^1T^{-2}
Vector	$\langle 1, 1, -2\rangle$	$\langle 1, 0, 0\rangle$	$\langle 0, 1, -2\rangle$

This exemplifies how multiplication of physical entities corresponds to "multiplication" of dimensions, which in turn corresponds to addition of dimension vectors.

1.2 Inference, Types, and Dimensions

Logical *inference systems* consist of a set of *inference rules*, with a number of *premises* and a *conclusion*. An example is the famous Modus Ponens rule from propositional logic:

$$\text{MP} : \frac{\mathcal{P} \quad \mathcal{P} \Rightarrow \mathcal{Q}}{\mathcal{Q}}$$

It states that "if $\mathcal{P}$ holds and if $\mathcal{P} \Rightarrow \mathcal{Q}$ holds, then we can deduce that $\mathcal{Q}$ holds".

Inference systems are used to formalize logics. If the logic is simple enough, then there might exist an *inference algorithm* that effectively finds a proof or refutal for a given statement.

Type Inference

Type systems for programming languages can be formulated as inference systems. A classical type system is *Hindley-Milner's type system* [9, 13]. It is formulated for a simple functional language, and has the ability not only to check types, but also to find sensible types for identifiers that are not type declared. Statements in this system has the form $\Gamma \vdash e : \sigma$, where e is an expression, σ is a type, and Γ is a binding of types to identifiers. It reads "if identifiers are typed according to Γ, then e has type σ". This conditional form of the statements gives the ability to find types for identifiers, since Γ is derived along with the typing of e. Hindley-Milner's type system has an efficient inference algorithm [5].

An example of an inference rule in the system is the one for *function application*:

$$\text{APP} : \frac{\Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash e : \sigma}{\Gamma \vdash f(e) : \tau}$$

It reads "if the function f has type $\sigma \to \tau$ and the argument e type σ, then the result $f(e)$ has type τ". Here, $\sigma \to \tau$ is a *function type*.

Hindley-Milner's type system is *polymorphic*, which means that identifiers can have many types. An example is the identity function *id* which is given the type $\forall\alpha.\alpha \to \alpha$ by the type system. Here, α is a *type variable*. Substituting different types for α gives the possible types for *id*, e.g., $id : int \to int$ and $id : string \to string$. Thus, $id(17) : int$ and $id(\texttt{"xyz"}) : string$.

Inference of Dimensions

Dimensional systems can be seen as type systems, where the "types" for expressions in equations are dimensions. Our tool is based on the polymorphic dimensional type system presented in Section 3.3, which is closely related to Hindley-Milner's system. A statement in this system has the form $\mathcal{C} \vdash e : \delta$, where $\mathcal{C}$ is a system of linear equations relating the "dimensional variables" for the identifiers in the equations.

An inference algorithm operates on the equations of the specification and uses the statements of the type system to infer new information in the form of dimensional equations. The dimensional equations are then collected in a system of linear equations which can be solved with standard linear system solving methods.

2. RELATED WORK

A lot of time and work has been put into the research of DA. Mostly this work has been concentrated on DA for general purpose programming languages, and not for specification languages. We will not give a full account for this here, due to space considerations, but try to focus on the most relevant achievements within the field.

2.1 Dimensional Analysis

DA is far from new in science. For centuries scientists have used DA to make an initial estimate of the correctness of their formulas and equations. Even the Π-theorem [3] uses DA as an underlying model.

One of the early references to DA and computer science is Thun [18], he shows a clear connection between DA and computer science, as well as showing some properties of dimensional spaces.

Many languages have been extended or augmented to incorporate DA, most of them are general programming languages, as opposed to specification languages for simulation of dynamic systems.

Hilfinger [8] and Rogers [16] presented DA for Ada. Agrawal et al [1] shows DA for Pascal. Umrigar [19] and Barton et al [2] presented a DA system for C++.

The specification language Z [17] has been extended with DA by Hayes [7]. This work is the most closely related to our work with respect to language support.

2.2 Dimensional Inference

Wand and O'Keefe [20] proposed a dimensional inference system for the functional language ML. Their system represents dimensions as vectors of rational exponents and infers dimensions by solving corresponding constraint equations using gaussian elimination.

Some work has been done to further DI, mainly by Kennedy [11]. Kennedy shows that there is a most general dimension [10] and how to infer it in ML. Kennedy goes further and presents a fundamental equation system [12] for inference and shows that it can be applied to dimensions as well.

3. IMPLEMENTATION FOR GPROMS

gPROMS is a statically typed programming language for simulation, optimization and parameter estimation of complex processes. The gPROMS simulation specifications are built using a simple syntax which allows for general mathematical equations to be formalized.

We have developed a *semi-polymorphic* dimensional type inference system for gPROMS that can be used to analyze the dimensional correctness of the simulation specifications as well as to infer missing dimensions. In our DI system, each variable and parameter is associated with a dimension type, much like the unit annotation of variable types in gPROMS, representing its dimension. The dimension types are then used to infer dimensions and derive dimensional consistency using the rules of the type system. In the following text we will refer to both variables and model parameters as symbol entities or symbols.

The concept of semi-polymorphism is based on the idea of treating certain constructs as polymorphic and other as *monomorphic*. The polymorphism enables polymorphic parameterized models to be defined, while the monomorphism is required to safely analyze the dimensional consistency of the model equations. A polymorphic model can work on a range of different dimensions. One such example would be a polymorphic regulator model. The same general regulator could be applied to control a flow as well as a position. However, once the general model is instantiated we need the quantities to be monomorphic in order to safely check the dimensional consistency of the actual equations involved. Without the monomorphism certain dimensional inconsistencies would be left undetected, since that would allow different occurrences of the same symbol to have different dimensions.

Basically our inference system transforms the equations that make up the simulation specification into equivalent dimensional constraint equations based on dimensional analysis theorems. The constraints form a system of potentially independent linear equations which are solved by means of linear algebra, or specifically Gauss-Jordan Row Reduction.

The solution to the system of equations is the inferred dimensions for the symbols in order for the system to be dimensionally consistent. The dimensional analysis is performed at the same time, since a dimensionally inconsistent system does not have any solutions.

There are three possible outcomes when solving the constraints; a single solution, an infinite number of solutions or no solutions at all. In case of no solutions, we have a dimensionally inconsistent system. If we have exactly one solution, the system is dimensionally consistent and each dimension is fully known. If we have infinite number of solutions, we have a dimensionally consistent system with dependencies between the unknown dimensions.

3.1 Dimensional Information

Without any information about the dimensions of the symbols, our DI system can only partially analyze the dimensional consistency of a simulation specification. In order for the system to derive the actual dimensions of the symbols, some static dimensional information is required. This information could be extracted from the unit annotations of variable types, but then we would still require some other annotation for the parameter declarations.

We have incorporated dimensional annotation in gPROMS for both variable types and model parameters. The dimensional annotations are placed within tagged block comments so the source program specification is still compatible with the gPROMS environment. The special annotation comment is on the following form: `{@dim d}`, with `d` replaced by the actual dimension.

We decided to use the same base dimensions as the SI system of units and their corresponding names as acronyms. This means that instead of using, for instance, the dimension *Length*, the SI unit m is used instead. The corresponding dimensional annotation would be `{@dim m}`.

3.2 Dimension Types

Each symbol is associated with a dimension variable ($\vec{d}$). A dimension is represented as a vector of rational numbers, where each position corresponds to an exponent of a base dimension as described in Section 1.1. A dimensionless quantity is represented with the zero vector ($\vec{0}$). Dimension types (δ) are built from linear combinations of dimension variables closed under a vector space.

The intrinsic functions are given polymorphic types represented by the type scheme: $\sigma_\delta = \forall \delta_1 . \delta_1 \rightarrow \delta_2$, which asserts that the functions will work properly for all possible dimensions. Type schemes are commonly used in polymorphic programming languages to support polymorphic types [4]. Polymorphism is obtained by allowing the type scheme to be instantiated with different bindings for the universally quantified types.

The trigonometric functions are treated as polymorphic functions with a dimensionless result. For instance, the trigonometric function `SIN` is assigned the following type scheme:

$$\texttt{SIN} : \forall \delta . (\delta \rightarrow \vec{0})$$

The following type scheme is assigned to the `SQRT` function, since our system is based on rational exponents:

$$\texttt{SQRT} : \forall \delta . (\delta \rightarrow \frac{1}{2}\delta)$$

Basically, the typing of `SQRT` states that the resulting dimensional expression is generated by multiplying the dimensional expression of the argument by the rational factor $\frac{1}{2}$.

The arithmetic operators are given the following polymorphic types:

$$\begin{array}{rcl} \texttt{+,-} & : & \forall \delta . (\delta \times \delta \rightarrow \delta) \\ \texttt{*} & : & \forall \delta_1 \delta_2 . (\delta_1 \times \delta_2 \rightarrow \delta_1 + \delta_2) \\ \texttt{/} & : & \forall \delta_1 \delta_2 . (\delta_1 \times \delta_2 \rightarrow \delta_1 - \delta_2) \end{array}$$

The type schemes for the operators are incorporated into the inference rules given in Section 3.3.

3.3 Dimension Type Rules

The language elements of gPROMS are associated with corresponding statements in our dimensional type system, called type rules, which are on the following form: $\mathcal{C} \vdash e : \delta$, where $\mathcal{C}$ is a set of linear equations, e is an expression, and δ a dimension type. It reads “if identifiers are dimensionally constrained by $\mathcal{C}$, expression e has dimension δ”.

Each equation represents an equivalence relation between two dimension types, effectively forcing them to represent the same dimension. For instance, two quantities that are added must have the same dimension. The purpose of the type rules is to derive such a system of equations, constraining the dimensions and thus ensuring dimensional consistency for the entire simulation specification.

Because of limited space we will only present the most important type rules. A complete set of rules is presented in [15]. These are the basic type rules:

$$\text{eqn} \quad : \quad \frac{\mathcal{C}_1 \vdash e_1 : \delta_1 \qquad \mathcal{C}_2 \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\delta_1 = \delta_2\} \vdash e_1 = e_2 : \delta_1}$$

$$\text{add} \quad : \quad \frac{\mathcal{C}_1 \vdash e_1 : \delta_1 \qquad \mathcal{C}_2 \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\delta_1 = \delta_2\} \vdash e_1 \texttt{ + } e_2 : \delta_1}$$

$$\text{sub} \quad : \quad \frac{\mathcal{C}_1 \vdash e_1 : \delta_1 \qquad \mathcal{C}_2 \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\delta_1 = \delta_2\} \vdash e_1 \texttt{ - } e_2 : \delta_1}$$

$$\text{mul} \quad : \quad \frac{\mathcal{C}_1 \vdash e_1 : \delta_1 \qquad \mathcal{C}_2 \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \vdash e_1 \texttt{ * } e_2 : \delta_1 + \delta_2}$$

$$\text{div} \quad : \quad \frac{\mathcal{C}_1 \vdash e_1 : \delta_1 \qquad \mathcal{C}_2 \vdash e_2 : \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \vdash e_1 \texttt{ / } e_2 : \delta_1 - \delta_2}$$

$$\text{timederiv} \quad : \quad \frac{\mathcal{C} \vdash e : \delta}{\mathcal{C} \vdash \texttt{\$}\, e : \delta - \langle 0, 0, 1 \rangle}$$

$$\text{app} \quad : \quad \frac{\mathcal{C}_1 \vdash e : \delta_1 \qquad \mathcal{C}_2 \vdash \mathit{func} : \delta_1 \to \delta_2}{\mathcal{C}_1 \cup \mathcal{C}_2 \vdash \mathit{func(e)} : \delta_2}$$

$$\text{spec} \quad : \quad \frac{\mathcal{C} \vdash \mathit{func} : \sigma_{\delta_1}}{\mathcal{C} \vdash \mathit{func} : \sigma_{\delta_1}[\delta/\delta_1]}$$

$$\text{id} \quad : \quad \frac{}{\emptyset \vdash \mathit{id} : \mathit{env}_{\vec{d}}(\mathit{id})}$$

In the type rules above, $\mathit{env}_{\vec{d}}$ is a static environment that maps symbols to their corresponding dimension variables and functions to their associated type schemes. Bindings in $\mathit{env}_{\vec{d}}$ are looked up in the following way: $\mathit{env}_{\vec{d}}(\mathit{id}) = \vec{d}$, where id is either a symbol or a function name.

Another environment, env_δ, contains mappings between dimension variables and dimensions. The dimension variables are either bound or unbound. If a symbol is dimensionally annotated, the corresponding dimension variable will be bound to the annotated dimension. Unbound variables are considered free and can thus be bound to any dimension. Once a variable is bound the binding cannot be changed which reflects the temporal invariance of dimensions.

3.4 Inference Algorithm

We will now give a description of our dimensional inference algorithm. These are the basic steps of our inference engine:

1. Derive system of equations, $\mathcal{C}$.
2. Infer trivial dimensions.
3. Solve systems of equations.

Derive system of equations

The system of equations $\mathcal{C}$ is derived using the type rules descried earlier. In practice, the gPROMS source specification is parsed and converted into an abstract syntax tree (AST) whereafter the type rules are applied recursively on the structure. The resulting system of equations contains all dependencies necessary to analyze the dimensional consistency.

Infer trivial dimensions

Before we solve the system of equations we use the statically annotated information to recursively infer new dimensions, by an extended back substitution algorithm. The process is based on finding equations with only one unknown, which are trivially solved.

When such an equation is found, the binding for the unknown dimension variable is updated and all occurrences of it are substituted for its dimension. Any resulting redundant equations are removed. This part of the algorithm terminates when there are no more trivial equations to solve.

During this step it is possible to detect dimensional inconsistencies simply by finding equations where the two sides are not dimensionally equivalent. Such a property is easily checked once all dimension variables in an equation are known.

One of the main reasons why we introduce this step is because we believe that given enough dimensional annotations, this step will reduce the overall execution time and at the same time offer qualitative dimensional inconsistency reports. The dimensional inconsistencies found during Gauss-Jordan Row Reduction are impossible to trace, since the structure of the system is destroyed during Row Reduction.

Solve systems of equations

If not all equations were removed in the previous step, the system of equations is divided into one or more partitions which can be solved independently. A partition is defined as the least system of equations satisfying the condition that each dimension variable must only occur in the equations of one specific partition.

We do not experience any of the numerical problems usually attributed to Gaussian elimination since we have based our computations on rational numbers.

Infinite number of solutions

As pointed out earlier, if the systems prove to be solvable the solution is the inferred dimensions in order for the simulation specification to be *dimensionally consistent*. Also, when there are no unknown dimensions we say that the simulation specification is *dimensionally complete*.

In the case of infinite number of solutions the specification is dimensionally consistent but it is not complete, since there are still unknown dimensions. Often the user might want to verify that the inferred dimensions are physically correct which is only possible if the specification is complete.

We have developed a heuristics that suggests to the user which variables/parameters that should be annotated in or-

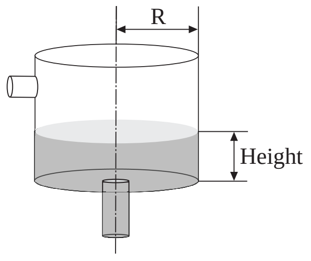

Figure 1: Schematics of a simple buffered tank

der for the simulation specification to be dimensionally complete as well as dimensionally consistent. Intuitively, the heuristics favors those symbols that are most frequently occurring.

3.5 DI Tool Description

Our DI system has been implemented in Java as a standalone tool in the form of a command-line based console application. The tool is invoked as a separate pass either during model development or before the actual simulation. The input is a gPROMS source simulation specification and the output states the corresponding dimensional consistency.

Polymorphic models can be specified if some variable types are not dimensionally annotated. Variable types that are not annotated are considered polymorphic, which ensures that variables of that type can have different dimensions. For instance, to specify a polymorphic regulator model at least the variable types of the input and output should not be annotated.

4. GPROMS MODEL EXAMPLE

Consider the simple gPROMS model of a buffered expansion tank depicted in figure 4 and the following gPROMS source specification:

```
DECLARE
  VARIABLE
    Length = 0 : 1E-10 : 1E10
    Flow   = 0 : 1E-10 : 1E10 {@dim m^3/s}
    Volume = 0 : 1E-10 : 1E10
END

MODEL BufferTank
  PARAMETER
    CorrFactor    AS REAL
    R             AS REAL

  VARIABLE
    Buffer        AS Volume
    Inlet, Outlet AS Flow
    Height        AS Length

  EQUATION
    $Buffer = Inlet - Outlet ;
    Buffer = 3.14159 * R^2 * Height ;
    Outlet = CorrFactor * SQRT( Height ) ;
END
```

The following constraint equations are generated from the source model specification:

$$\begin{aligned}
\vec{d}_{Buffer} - \vec{d}_{Flow} &= \langle 0,0,1\rangle \\
\vec{d}_{Flow} - \vec{d}_{Flow} &= \langle 0,0,0\rangle \\
\vec{d}_{Buffer} - 2\vec{d}_{R} - \vec{d}_{Height} &= \langle 0,0,0\rangle \\
\vec{d}_{Flow} - \vec{d}_{CorrFactor} - \tfrac{1}{2}\vec{d}_{Height} &= \langle 0,0,0\rangle
\end{aligned}$$

In the above equations, each dimension variable corresponds to a specific symbol in the source model. For instance, the associated dimension variable of the parameter `R` is $\vec{d}_R$. Observe that the variables `Inlet` and `Outlet` are replaced by the associated dimension variable of their type, $\vec{d}_{Flow}$, since it is statically annotated and no longer polymorphic. The equation $\vec{d}_{Flow} - \vec{d}_{Flow} = \langle 0,0,0\rangle$ is redundant and thus removed. The DI engine now applies the dimensions known via dimensional annotations in order to infer new dimensions recursively. Since the symbol `Flow` is known to be of dimension `m^3/s` all occurrences of `Flow` are substituted for its dimension, which leaves us with the following equations:

$$\begin{aligned}
\vec{d}_{Buffer} &= \langle 0,3,0\rangle \\
\vec{d}_{Buffer} - 2\vec{d}_{R} - \vec{d}_{Height} &= \langle 0,0,0\rangle \\
-\vec{d}_{CorrFactor} - \tfrac{1}{2}\vec{d}_{Height} &= \langle 0,-3,1\rangle
\end{aligned}$$

According to the equation $\vec{d}_{Buffer} = \langle 0,3,0\rangle$, $\vec{d}_{Buffer}$ must be bound to $\langle 0,3,0\rangle$. Now all occurrences of $\vec{d}_{Buffer}$ are substituted for $\langle 0,3,0\rangle$ and the equation is removed, yielding the following equations:

$$\begin{aligned}
-2\vec{d}_{R} - \vec{d}_{Height} &= \langle 0,-3,0\rangle \\
-\vec{d}_{CorrFactor} - \tfrac{1}{2}\vec{d}_{Height} &= \langle 0,-3,1\rangle
\end{aligned}$$

The corresponding system of equations proved to be consistent, but since we only have two equations but three unknowns the system could not be unambiguously solved. Therefore the inference system given our heuristics suggest that the symbol `Length` should be annotated (the variable type of `Height`). If we annotate `Length` to be of dimension m and run the analysis again, we end up with the following inferred dimensions:

$$\begin{aligned}
\vec{d}_{Buffer} &= \langle 0,3,0\rangle \\
\vec{d}_{R} &= \langle 0,1,0\rangle \\
\vec{d}_{CorrFactor} &= \langle 0,\tfrac{5}{2},-1\rangle \\
\vec{d}_{Flow} &= \langle 0,3,-1\rangle \\
\vec{d}_{Height} &= \langle 0,1,0\rangle
\end{aligned}$$

As a result, the simulation specification was proven to be dimensionally consistent as well as complete. Due to the completeness the simulationist can now analyze the inferred dimensions and verify their correctness.

The inferred dimension for `CorrFactor` is possible since we use rational exponents. If we would have used integer exponents, like Kennedy [11], the inference would have failed.

5. CONCLUSIONS

We have presented an analysis method to guarantee the dimensional correctness for simulation specification languages. Further, we have implemented a prototype for gPROMS, and another tool for Modelica is under development.

We strongly believe that Dimensional Inference can increase correctness and the quality of simulations by insuring that the models are dimensionally correct.

Our gPROMS tool is ready for extensive testing of commercial models. We have performed small and medium test cases and are waiting on the Modelica tool to be finished so we can continue with extensive testing of the Modelica library.

6. FUTURE WORK

As we stated before our analysis method has been implemented as a stand-alone tool for gPROMS and will also be soon for Modelica. This is not an optimal solution in our opinion. Our method should be integrated into the simulation language tools. We also believe that our analysis method could be more effective if the specification languages are extended to incorporate dimensional constructs. So far we have used comments in gPROMS to annotate dimensional information, which is cumbersome and time-consuming.

The next natural step in the research would be to extend DI to be a full unit checking system. A unit checking system would be able to dimensionally check models developed in the English Imperial measurement system in conjunction with the SI system of measurement, thereby helping to increase the correctness for such models.

We would like to see our analysis method used in more simulation specification languages as well as other simulation environments that might not use a language as a primary specification model (e.g. visual specification of models).

7. ACKNOWLEDGEMENTS

We would like to thank ABB Corporate Research Center, Department of Industrial IT and the KK-foundation (grant 2001/0214) for their support.

8. REFERENCES

[1] M. B. Agrawal and V. K. Garg. Dimensional analysis in PASCAL. *ACM SIGPLAN Notices*, 19(3):7–11, Mar. 1984.

[2] J. J. Barton and L. R. Nackman. Dimensional analysis. *C++ Report*, 7(1):39–40, 42–43, Jan. 1995.

[3] E. Buckingham. On physically similar systems: Illustrations of the use of dimensional equations. *Phys. Rev. 4*, pages 345–376, 1914.

[4] L. Cardelli. Basic polymorphic typechecking. Computing Science Technical Report 112, AT&T Bell Laboratories, Murray Hill, 1984.

[5] L. Damas and R. Milner. Principal type-schemes for functional programs. pages 207–212, 1982.

[6] P. Fritzson and V. Engelson. Modelica—A unified object-oriented language for system modeling and simulation. In E. Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 1998.

[7] I. Hayes and B. Mahony. Using Units of Measurement in Formal Specifications. *Formal Aspects of Computing*, 7(3):329–347, 1995.

[8] P. N. Hilfinger. An ada package for dimensional analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, Apr. 1988.

[9] J. R. Hindley. The principal type scheme of an object in combinatory logic. volume 146, pages 29–60, 1969.

[10] A. J. Kennedy. Dimension types. In D. Sannella, editor, *Programming Languages and Systems—ESOP'94, 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362, Edinburgh, U.K., 11–13 Apr. 1994. Springer-Verlag (LNCS 788).

[11] A. J. Kennedy. *Programming Languages and Dimensions.* PhD thesis, University of Cambridge, Computer Laboratory, Cambridge, United Kingdom, April 1996.

[12] A. J. Kennedy. Type inference and equational theories. Technical Report LIX/RR/96/09, LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France, Sept. 1996.

[13] R. Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[14] C. Pantelides. An advanced tool for process modelling, simulation and optimisation. In *CHEMPUTERS EUROPA III, Frankfurt*, 1996.

[15] D. Persson. Dimensional inference for gproms. Master's thesis, Mälardalen University, Computer Science Laboratory, Västerås, Sweden, in preparation. 2003.

[16] P. Rogers. Dimensional analysis in Ada. *ACM SIGADA Ada Letters*, 8(5):92–100, Sept./Oct. 1988.

[17] J. M. Spivey. *The Z Notation : a reference Manual.* C.A.R. Hoare Series Editor, 1989.

[18] R. E. Thun. On dimensional analysis. *IBM Journal of Research and Development*, 4:349–356, 1960.

[19] Z. D. Umrigar. Fully static dimensional analysis with C++. *ACM SIGPLAN Notices*, 29(9):135–139, Sept. 1994.

[20] M. Wand and P. M. O'Keefe. Automatic dimensional inference. *Computational Logic: in honor of J. Alan Robinson*, pages 479–486, 1991.