Automatic Dimensional Analysis of Cyber-Physical Systems

Conference Paper · August 2012

DOI: 10.1007/978-3-642-32759-9_30

3 authors:

Sam Owre
SRI International
76 PUBLICATIONS **5,356** CITATIONS
SEE PROFILE

Indranil Saha
Indian Institute of Technology Kanpur
35 PUBLICATIONS **848** CITATIONS
SEE PROFILE

Natarajan Shankar
SRI International
196 PUBLICATIONS **8,026** CITATIONS
SEE PROFILE

Automatic Dimensional Analysis of Cyber-Physical Systems*

Sam Owre[1], Indranil Saha[2], and Natarajan Shankar[1]

[1] Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA
[2] Computer Science Department, UCLA, CA 90095, USA
owre@csl.sri.com, indranil@cs.ucla.edu, shankar@csl.sri.com

Abstract. The first step in building a cyber-physical system is the construction of a faithful model that captures the relevant behaviors. Dimensional consistency provides the first check on the correctness of such models and the physical quantities represented in it. Though manual analysis of dimensions is used in physical sciences to find errors in formulas, this approach does not scale to complex cyber-physical systems with many interacting components. We present DimSim, a tool to automatically check the dimensional consistency of a cyber-physical system modeled in Simulink. DimSim generates a set of constraints from the Simulink model for each subsystem in a modular way, and solves them using the Gauss-Jordan elimination method. The tool depends on user-provided dimension annotations, and it can detect both inconsistency and underspecification in the given dimensional constraints. In case of a dimensional inconsistency, DimSim can provide a minimal set of constraints that captures the cause of the inconsistency. We have applied DimSim to numerous examples from different embedded system domains. Experimental results show that the dimensional analysis in DimSim is scalable and is capable of uncovering critical errors in models of cyber-physical systems.

Keywords: Cyber-Physical Systems, Simulink, Dimensional Analysis, Gauss-Jordan Elimination, Unsatisfiable Core

1 Introduction

Cyber-physical systems are complex computing systems that interact with physical processes. As the physical processes are closely coupled with the system, it cannot be developed without keeping the physical process in the loop. Thus the first step in building a cyber-physical system is the construction of a faithful model that captures the relevant behaviors. Physical quantities have associated dimensions that can be represented in terms of some set of base dimensions, for example, force can be given a dimension $mass \times length/(time \times time)$, where

* This work was supported by NSF Grant CSR-EHCS(CPS)-0834810 and NASA Cooperative Agreement NNX08AY53A. We received useful feedback from our colleagues Bruno Dutertre and Ashish Tiwari and from Professor Martin Hofmann of LMU Munich. We are especially grateful to the anonymous reviewers for their constructive and insightful feedback.

length, *mass* and *time* are base dimensions. Dimensions can be further classified into units so that length can be measured in inches or metres and mass in pounds or kilograms. Dimensions are used in the physical sciences to check the feasibility of computative formulas for physical laws for dimensional consistency [13]. Such laws also satisfy *dimensional invariance* so that they hold even under changes of units through scaling. Dimensions also provide a heuristic for suggesting such laws. Finally, dimensional analysis can be used to refactor a law involving n variables with dimensions built from d base dimensions in terms of $n - d$ dimensionless product terms.

As cyber-physical systems deal with physical processes, the variables associated with the model of a system often represent the numerical values of physical quantities. While constructing the model of a cyber-physical system, many common errors are indicated by mismatches in dimensions. Dimensional consistency provides the first check on the correctness of such models and the physical quantities represented in it. Manual analysis of dimensions is used in physical science to find errors, but this approach does not scale to complex cyber-physical systems with many interacting components. We present DimSim, a tool to automatically check dimensional consistency of a cyber-physical system modeled in Simulink. DimSim relies on user-provided dimension annotations for a small subset of the variables in the model. As type discipline plays a fundamental role in writing software programs, dimension discipline can play the same role for the design of cyber-physical systems where each variable has a physical meaning attached to it. We argue that dimension discipline eliminates a common source of errors in designing cyber-physical systems. The techniques used in DimSim are general and can be used for languages other than Simulink.

DimSim generates a set of constraints on the dimensions of the inputs and the outputs of the system as given by the interconnection of the system components. Depending on the provided dimension annotations, the dimensions of all the variables may or may not be uniquely determined. If the dimensions of all the variables can be uniquely determined, the dimensional safety question is reduced to the unique satisfiability question. Though the unique satisfiability problem is in general NP-hard [19], dimensional analysis does not have disjunctive constraints, and the resulting constraints can be solved in polynomial time. DimSim uses the Gauss-Jordan Elimination method [16] to solve the constraints and infer (possibly polymorphic) dimensions for all the variables. If the dimensions cannot be determined uniquely (modulo the dimensions of the external, i.e., input and output, variables of the block under analysis), the unification algorithm finds the most general dimension assignment for the variables. If the set of dimensional constraints is found to be inconsistent, DimSim provides a minimal subset of constraints that helps pinpoint the source of the error.

One of our goals is to provide a tool that is scalable to large models. As the Gauss-Jordan Elimination method takes cubic time to solve the constraints, we cannot solve the constraints generated from large models using this technique in reasonable time. To make DimSim scalable to large systems, we adopt a compositional strategy. In Simulink, one can model a cyber-physical system in a modular way by using subsystems. DimSim analyzes one subsystem at a time, and the constraints on the inputs and outputs of a subsystem are propagated

to its parent subsystem. While analyzing a higher level subsystem DimSim looks only at the dimensional constraints of the inputs and outputs of each component subsystem.

We have applied DimSim to a number of examples including a house heating model, a collision detection algorithm from the aerospace domain, and a number of automotive control systems. Our results show that DimSim is scalable and is capable of uncovering critical errors in the model.

In Section 2, we introduce the basic concepts of Simulink through an example, and provide the problem statements. In Section 3, we present our dimension analysis technique: we describe our dimensional constraint solver, and how the solver detects dimensional inconsistency and underspecification of dimensional annotations. In Section 4, we present experimental results on numerous examples. In Section 5 we outline the related work, and compare and contrast our approach.

2 Example and Problem Definitions

In this section, we introduce the basic terminology that is used in the rest of the paper, and formally define our problem.

2.1 Simulink Model

Simulink is used to model cyber-physical systems. A Simulink model of a cyber-physical system is composed of a number of blocks connected by wires. A block may be an elementary block that does not contain any other block, or it may be a subsystem that is composed of a number of elementary blocks and subsystems. Each occurrence of a block in a subsystem has a unique name.

Example. Figure 1 depicts the Simulink model of a cruise control system [3]. The objective of this control system is to maintain the speed of the car at a reference point. Thus the input to the model is the *reference speed* and the output is the *actual speed* of the vehicle. The model has two main subsystems: the *Controller* subsystem generates the control signal depending on the reference speed and the actual speed of the car, and the *VehicleDynamics* subsystem models the response of the vehicle to the control inputs.

We now introduce a few terms that are used in describing the dimensional analysis.

Source Block A Simulink block that does not have an input port is a Source block. For example, in Figure 1, *ReferenceSpeed* is a source block.

Sink Block A Simulink block that does not have an output port is called a Sink block. For example, *VehicleSpeed* is a sink block.

Value Parameter If a Simulink block requires an external value as a parameter, the parameter is called a Value parameter. For example, the constant in *VehicleMass* is a value parameter.

Port Variables As we explain later, the dimensions of the input and the output ports are represented by dimensional variables. The dimension variables

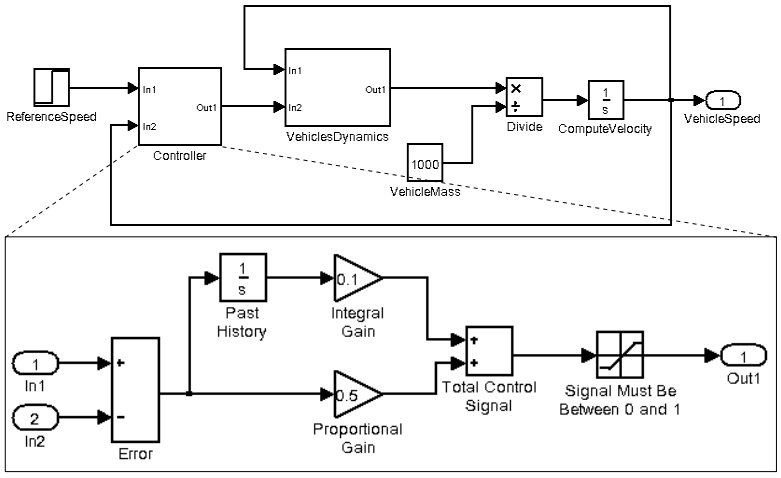

Fig. 1. Simulink model of a cruise control system

corresponding to the input and output ports of a block are named with suffixes, e.g., *__In1* and *__Out1*. For example, the variables corresponding to the ports of the *Divide* block are *Divide__In1*, *Divide__In2* and *Divide__Out1*, respectively.

Interface Variables This set of variables covers the dimensions of the output ports of the Source blocks, the input ports of the Sink blocks, and the variables used to hold the parameter values. The user is expected to provide dimension annotations for some of these variables.

External Variables The variables representing the dimensions of the input and output ports of a subsystem in a model are called external variables for the subsystem. For example, in Figure 1, *In1__Out1* (i.e., the output of the source block *In1*), *In2__Out1* and *Out1__In1* are external variables in the *Controller* subsystem. Note that the set of external variables of a subsystem is a subset of its interface variables.

Internal Variables In a subsystem, all port variables that are not external variables are internal variables. For example, in Figure 1, *Error__Out1* is an example of internal variable in the *Controller* subsystem.

2.2 Dimensional Constraints

Each port in a block in a Simulink subsystem is assigned a dimension variable. A dimension variable denotes a vector of rational numbers, where each position corresponds to an exponent of a base dimension. For example, if the base dimensions are length, mass, and time, then the dimension for force will be a vector $\langle L = 1, M = 1, T = -2 \rangle$. Note, however, that the set of base dimensions can be defined for each specific dimensional analysis problem.

In Table 1, we present the dimensional constraints generated from some basic Simulink blocks. DimSim handles many other Simulink blocks that are used in different embedded system domains. The dimension type of a Simulink block is a

Simulink Block	**Block Type**	**Constraints**
Abs, Unary Minus	$[D_1 \rightarrow D_2]$	$D_2 = D_1$
Add, Subtract	$[D_1,\ D_2 \rightarrow D_3]$	$D_2 = D_1,\ D_3 = D_1$
Product	$[D_1,\ D_2 \rightarrow D_3]$	$D_3 = add\_dim(D_1, D_2)$
Divide	$[D_1,\ D_2 \rightarrow D_3]$	$D_3 = add\_dim(D_1, inv\_dim(D_2))$
Trigonometric Function	$[D_1 \rightarrow D_2]$	$extract\_dim(D_1, \delta) = \begin{cases} 1 \text{ if } \delta = angle \\ 0 \text{ if } \delta \neq angle \end{cases}$ $D_2 = dimensionless$
Relational Operator	$[D_1,\ D_2 \rightarrow D_3]$	$D_2 = D_1,\ D_3 = dimensionless$
Memory, Unit Delay	$[D_1 \rightarrow D_2]$	$D_2 = D_1$
Integrator	$[D_1 \rightarrow D_2]$	$D_2 = inc\_dim(D_1,\ time,\ 1)$

Table 1. Dimensional constraints for different Simulink blocks

relation between the dimension variables representing the input ports of the block and the dimension variables representing the output ports. The function *add_dim* takes two dimension vectors as inputs and computes the output dimension vector element-wise by adding the exponents of the base dimensions of the inputs. The function *inv_dim* takes a dimension vector as input and returns a dimension vector whose components are obtained by negating the components of the input vector. The function *extract_dim* takes as argument, a dimension vector D and a base dimension δ, and returns the exponent value of the base dimension δ in the dimension vector D. The function *inc_dim* takes as input a dimension vector D_1, a base dimension δ which appears in D_1 and an integer k and returns a dimension vector D_2 which is obtained by adding k to the exponent of δ in D_1, and leaving the other base dimensions in D_1 unchanged.

As an example, the dimensional constraints from the cruise control model are shown in Figure 2.

2.3 Problem Definition

A user may provide a dimension annotation for an interface variable of a Simulink block. For example, the dimension of the source *ReferenceSpeed* is $\langle L = 1, M = 0, T = -1\rangle$ and the dimension of the output of the Constant block *VehicleMass* is $\langle L = 0, M = 1, T = 0\rangle$, where L, M, and T denote the exponent of the base dimensions *length*, *mass*, and *time*, respectively. We do not use a fixed set of base dimensions in DimSim, but instead extract the set of base dimensions from those used in the model.

Dimensional Safety. For a Simulink subsystem, if the port variables of the blocks can be assigned dimensions without violating any dimensional constraint for the basic Simulink blocks in the model, then the Simulink model is dimensionally safe.

DimSim solves the following three problems related to dimension analysis:

Problem 1. Given a Simulink model, find out if it is dimensionally safe, i.e., is there any solution to the set of dimensional constraints. If there is a solution, is there a compact way to represent the set of all solutions. Furthermore, determine

$(\texttt{VehicleDynamics\_\_In1} = \langle \texttt{L} = 1,\ \texttt{M} = 0,\ \texttt{T} = -1\rangle)\ \wedge (\texttt{VehicleDynamics\_\_In2} =$
$\langle \texttt{L} = 0,\ \texttt{M} = 0,\ \texttt{T} = 0\rangle)\ \wedge (\texttt{VehicleDynamics\_\_Out1} = \langle \texttt{L} = 1,\ \texttt{M} = 1,\ \texttt{T} = -2\rangle) \wedge$
(constraints from VehicleDynamics Subsystem)
$(\texttt{Controller\_\_In1} = \texttt{Controller\_\_In2})\ \wedge (\texttt{Controller\_\_Out1} = \langle \texttt{L} = 0,\ \texttt{M} = 0,\ \texttt{T} = 0\rangle) \wedge$
(constraints from Controller Subsystem)
$(\texttt{ReferenceSpeed\_\_Out1} = \langle\ \texttt{L} = 1,\ \texttt{M} = 0,\ \texttt{T} = -1\rangle)\ \wedge (\texttt{VehicleMass\_\_Out1} =$
$\langle\ \texttt{L} = 0,\ \texttt{M} = 1,\ \texttt{T} = 0\rangle)\ \wedge (\texttt{VehicleSpeed\_\_In1} = \langle\ \texttt{L} = 1,\ \texttt{M} = 0,\ \texttt{T} = -1\rangle) \wedge$
(constraints provided for interface variables)
$(\texttt{Controller\_\_In1} = \texttt{ReferenceSpeed\_\_Out1}) \wedge$
$(\texttt{Controller\_\_In2} = \texttt{ComputeVelocity\_\_Out1}) \wedge$
$(\texttt{VehicleDynamics\_\_In1} = \texttt{ComputeVelocity\_\_Out1}) \wedge$
$(\texttt{VehicleDynamics\_\_In2} = \texttt{Controller\_\_Out1}) \wedge$
$(\texttt{Divide\_\_In1} = \texttt{VehicleDynamics\_\_Out1})\ \wedge (\texttt{Divide\_\_In2} = \texttt{VehicleMass\_\_Out1}) \wedge$
$(\texttt{Divide\_\_Out1} = \mathit{add\_dim}(\texttt{Divide\_\_In1},\ \mathit{inv\_dim}(\texttt{Divide\_\_In2}))) \wedge$
$(\texttt{ComputeVelocity\_\_Out1} = \mathit{inc\_dim}(\texttt{Divide\_\_Out1}, \texttt{time}, 1)) \wedge$
$(\texttt{Out1\_\_In1} = \texttt{ComputeVelocity\_\_Out1})$
(constraints for polymorphic blocks)

Fig. 2. Dimensional constraints generated from Cruise Control model

the unique solution if there is one.

Problem 2. When a subsystem in a Simulink model is not dimensionally safe, i.e., there is no solution to the dimensional constraints, then provide a succinct explanation of the inconsistency.

Problem 3. When the dimensions of all the variables cannot be uniquely determined from the user provided annotations, compute a minimal set of variables for which the user should provide dimension annotations to determine the dimension of all the variables uniquely. We omit the details of the solution in order to focus on the solutions to the first two problems.

3 Dimension Analysis through Constraint Solving

Our objective is to find out if a Simulink model is dimensionally safe, by which we mean that it is possible to assign dimensions to all the variables in the Simulink model without any conflict. Our dimension analysis algorithm is modular and is executed on a Simulink model in a bottom-up manner. Dimensional safety analysis of a subsystem is performed when all the subsystems inside it have already been analyzed.

DimSim accepts a Simulink model annotated by dimensions. The required constraints are generated by a static analysis of the model, and the generated constraints are solved using Gauss-Jordan Elimination. There may be three outcomes of solving the constraints for a Simulink subsystem - there is no solution, there is a unique solution, and there is an infinite number of solutions. In case

there is no solution, the Simulink subsystem is dimensionally inconsistent, and we present to the user a minimal subset of constraints that explain the inconsistency. In case there is a unique solution the values of all dimension variables are known. And in case there are an infinite number of solutions, the Simulink subsystem is dimensionally consistent, but the dimensions of some variables are known only in terms of other dimension variables. Our objective is to determine if there is a unique dimension assignment for all the ports in a subsystem (modulo the dimensions of the external variables). We provide the user with a minimal set of dimension variables that should be annotated for obtaining such a unique dimension assignment of all the variables in the subsystem.

Constraints are solved in a modular manner so that each subsystem is analyzed exactly once using the externally visible constraints exported by each of its component subsystems. The solver also indicates if the dimensional assignment for the signals in a subsystem is unique relative to the dimensional assignment for the external variables of the subsystem.

A static analysis of each subsystem yields a set of constraints (as shown in Section 2.2) over the external and internal variables of the subsystem. For example, a subsystem b of the form $z = u + w; u = xy$ with output variable z, a set of input variables $\{x, y, w\}$, and internal variable u, yields the constraints $Z = U = W; U = X + Y$, where X is a *dimension variable*. Here, each dimension variable X represents a vector $\langle x_1, \ldots, x_d \rangle$ for base dimensions $\delta_1, \ldots, \delta_d$, where x_i is the value for x on the dimension δ_i. The dimension solver transforms these constraints on dimension variables into a reduced row-echelon form. The dimension variables are ordered so that an external dimension variable, i.e., a dimension variable X corresponding to an external variable x, is never solved in terms of an internal dimension variable, i.e., a dimension variable Y corresponding to an internal variable y. In the above example, this yields the solved form $U = W, Z = W, X = W - Y$. This solved form also shows that there is a unique solution to the dimensional constraints modulo the assignment of dimensions to the external variables since it contains no internal non-basic (i.e., free) dimension variables. We can then project out the solved form on the external dimension variables to obtain $Z = W, X = W - Y$. This projected set of constraints, suitably renamed in terms of port variables, is exported to the subsystems that use b. If, for example, the dimension of w in the parent subsystem is identified as $\langle L = 1, M = 1, T = 0 \rangle$, and the dimension of x is identified as $\langle L = 1, M = 1, T = -1 \rangle$, then we can infer that the dimension of y is $\langle L = 0, M = 0, T = 1 \rangle$ and the dimension of u and z is $\langle L = 1, M = 1, T = 0 \rangle$. We represent constant dimension vectors with labels L, M, and T for mnemonic convenience, but a vector $\langle L = 1, M = 1, T = 0 \rangle$ would just be $\langle 1, 1, 0 \rangle$, if the dimensions are length, mass, and time, in that order. Also, the vector $\langle L = 0, M = 0, T = 0 \rangle$ is written as 0.

A dimensional inconsistency can arise when an expression such as $x + \int x\, dt$ generates the unsolvable constraint $X = X + \langle L = 0, M = 0, T = 1 \rangle$. A set of constraints is underspecified if the dimension of some internal variable of a subsystem is not uniquely determined by the dimensions of its external variables. Dimensional underspecification can be ruled out for *well-formed* systems where each value parameter has a given dimension, and each delay element is initialized

by such a value parameter with an associated dimension. Underspecification can arise in the absence of well-formedness. For example, suppose a subsystem with input variables x and y and output variable z defines internal variables u and v such that initially, $u_0 = v_0 = 0$, and $u' = u + ux/(x+y)$ and $v' = v + vy/(x-y)$, and $z = u/(1+v)$, where u' and v' denote the values of the internal variables u and v in the next time step. Since 0 is a dimensionally polymorphic constant, this yields the dimensional constraints $Z = U - V; X = Y$, and there is no way to determine the values of U and V from the dimensional assignment to x, y, and z.

We describe below the details of our modular analysis where we solve the constraints for each subsystem in terms of the solutions provided by analyzing its component subsystems. The dimension solver maintains a table T that contains an entry for each subsystem b in the design that records its

1. Set of external variables Θ_b
2. Set of internal variables Υ_b (disjoint from Θ_b)
3. An array S_b of reduced row-echelon solved forms

For each subsystem b, the solver takes as input the external variables Θ_b, internal variables Υ_b, the constraints Γ_b, and the imported subsystems $b_1\rho_1, \ldots, b_N\rho_N$, where each ρ_i is the wiring that maps the external variables of b_i to $\Theta_b \cup \Upsilon_b$. Each constraint in Γ_b is a sum of monomials, where each monomial is either of the form $k_i X_i$ for some rational coefficient k_i and $X_i \in \Theta_b \cup \Upsilon_b$ or of the form $k\langle L = \mathbf{l}, M = \mathbf{m}, T = \mathbf{t}\rangle$ for rational constants k, $\mathbf{l}$, $\mathbf{m}$, and $\mathbf{t}$. For example $X - 2Y - \langle L = 1, M = 0, T = -2\rangle$ is a possible constraint. The interpretation is that this represents the condition $X - 2Y - \langle L = 1, M = 0, T = -2\rangle = \langle L = 0, M = 0, T = 0\rangle$, where the summation operation $X + Y$ represents vector addition, and kX represents scaling. The constraints imported from each block b_i are renamed using ρ_i so as to map the external variables of b_i to the variables of b. For example, an adder block of the form $z = x + y$ might be used in a larger block with the x and y inputs renamed as u and the output z renamed as v. In this case ρ will be $\{x \mapsto u, y \mapsto u, z \mapsto v\}$.

The modular solver builds S_b by computing $solve(\Theta_b, \Upsilon_b, \Gamma_b \cup \bigcup_{i=1}^{N} \rho_i(\hat{S}_{b_i}))$. Here, Γ_b represents the constraints from subsystem b on the variables in $\Theta_b \cup \Upsilon_b$, $\hat{S}_{b_i}$ is the set of equations exported from S_{b_i}, and $\rho_i(\hat{S}_{b_i})$ is the result of renaming the equations $\hat{S}_{b_i}$ using the map ρ_i. If $\rho_i(x) = z$, then correspondingly, $\rho_i(X) = Z$ on the dimensional variables. In placing these constraints into a reduced row-echelon solved form, the variables are ordered so that internal variables are basic variables in preference to external variables. This is done by numbering the variables so that external variables are assigned smaller numbers than internal ones, and solving each equation in terms of its largest variable. The solving process can fail signalling an inconsistency when, for example, an equation of the form $v = 0$ is introduced, where v is a non-zero vector, i.e., of the form $\langle L = \mathbf{l}, M = \mathbf{m}, T = \mathbf{t}\rangle$ where $\mathbf{l}$, $\mathbf{m}$, or $\mathbf{t}$ is non-zero. In this case, the solver returns $\perp$. If we have a non-$\perp$ solved form S_b the exported constraints $\hat{S}_b$ are those constraints of S_b in the external variables Θ_b.

We use a simple incremental Gauss–Jordan solver for building S_b. For a set of constraints over $\Theta_b \cup \Upsilon_b$, we order the variables so that those in Θ_b are

below those in Υ_b. For example, if $\Theta_b = \{x, w\}$ and $\Upsilon_b = \{y, z, u\}$, then we can order the variables as $X < W < Y < Z < U$. We assume that all linear polynomials are always represented as ordered sum of monomials, where each monomial is either of the form kX for a nonzero rational coefficient k or is a constant monomial kv with a nonzero k and a vector constant v, so that k_1X_1 precedes k_2X_2 iff $X_1 > X_2$, and the constant monomial always occurs last. For example, if a polynomial is of the form $(2Z - \langle L = 1, M = 0, T = -2\rangle + 3X - 4W) - (5Y - 2U + 4Z)$, its ordered sum of monomials has the form $3X + (-4)W + (-5)Y + (-2)Z + 2U + \langle L = 1, M = 0, T = -2\rangle$.

The input set of polynomial constraints have the form $\{p_0 = 0, \ldots, p_n = 0\}$. The solved form is a polynomial constraint of the form $\{g_0 = 0, \ldots, g_m = 0\}$, where the leading monomial in each polynomial g_i has a distinct variable X_i, i.e., the basic variable in g_i, so that $X_i \not\equiv X_j$ for $i \neq j$, and no X_i occurs in g_j for $i \neq j$. If Ψ is a solved form, then $\Psi(p)$ represents the result of replacing each occurrence X_i in p by g_i'/k_i, where X_i is a basic variable in a polynomial $g_i = 0$ in Ψ of the form $k_iX_i + g_i'$. This substitution operation can be extended to sets of polynomials so that $\Psi(\Pi)$ is just the image of Π with respect to Ψ. The *solve* procedure can be defined as follows.

$$solve(\Theta, \Upsilon, \Pi) = incsolve(\Pi, \emptyset)$$

$$incsolve(\{p\} \cup \Pi', \Psi) = \begin{cases} \bot, \text{if } p' \equiv k, \text{ for } k \neq 0 \\ incsolve(\Pi', \Psi), \text{if } p' \equiv k, \text{ for } k = 0 \\ incsolve(\Pi', \{p'\}(\Psi) \cup \{p'\}) \end{cases}$$
$$\text{where } p' \equiv \Psi(p)$$

In this procedure, the initial solved form is empty, and hence there are no basic variables. If we have processed i input constraints, then we have a state Ψ_i to which we add the input constraint $p_{i+1} = 0$. We first place p_{i+1} in an ordered sum of monomials form with monomials ordered in decreasing order of their variables with the constant monomial placed last. We then obtain p'_{i+1} as $\Psi_i(p_{i+1})$, the result of substituting each basic variable X of Ψ_i in p_{i+1} by $\Psi_i(X)$ and placing the result in an ordered sum of products form. Note that all the variables in p'_{i+1} are non-basic in Ψ_i. If the resulting p'_{i+1} is just a constant vector v, then either $v = 0$, the equation is redundant, or $v \neq 0$ and we have an inconsistency and $\Psi_{i+1} = \bot$. Otherwise, p'_{i+1} has the form $kX + p''$ with X as the maximal variable, then we obtain Ψ_{i+1} by adding $p'_{i+1} = 0$ to Ψ_i and replacing each occurrence of X in Ψ_i by $-p''/k$ so that X is now a basic variable in Ψ_{i+1}.

For example, with Θ_b, Υ_b as before, if we start with an empty solution set Ψ_0, we can add $X + 2Y - U = 0$ by normalizing it as $U - X - 2Y = 0$ and adding it to Ψ_0 to obtain Ψ_1. Next, we can add $3W - 2U + X = 0$ by normalizing and substituting the solution for U to get $3W - 2(X + 2Y) + X = 0$. This is normalized as $4Y - 3W + X = 0$ which is added to Ψ_1 after replacing each occurrence of Y in g_1 by $\frac{3}{4}W - \frac{1}{4}X$ to obtain $U - \frac{3}{2}W - \frac{1}{2}X$ so that Ψ_2 is $\{U - \frac{3}{2}W - \frac{1}{2}X, Y - \frac{3}{4}W + \frac{1}{4}X\}$. If there are n constraints in Γ_b, then either $S_b = \bot$ if $\Psi_i = \bot$ for some i, or $S_b = \Psi_n$.

We can now sketch the argument for the correctness of the solver. The dimensional constraints of b are given by $\Gamma_b \wedge \bigwedge_{i=1}^{N} \exists \Upsilon_{b_i}.\rho_i(\Gamma_{b_i})$, where $b_1\rho_1, \ldots, b_N\rho_N$ are the subsystems appearing in b, and $\rho_i(\Gamma_{b_i})$ renames each external variable X of Γ_{b_i} to $\rho_i(X)$.

Proposition 1. *incsolve*$(\Pi, \emptyset) = \bot$, *iff the set of constraints* Π *is unsatisfiable.*

It can be easily checked that in each step of the *incsolve* procedure going from Π, Ψ to Π', Ψ', every solution of Π, Ψ, i.e., an assignment of dimensional vector values to variables in Π, Ψ, is a solution for Π', Ψ', and vice versa. In particular, note that every variable in Π, Ψ also occurs in Π', Ψ', and vice versa.

Proposition 2. *For a subsystem* b_i*, any assignment satisfying* $\hat{S}_{b_i}$ *can be extended to an assignment satisfying* S_{b_i}*.*

This is because the non-basic internal dimension variables in S_{b_i} can be freely assigned any value, e.g., the vector 0, and the assignments for the basic internal dimension variables are computed from those of the external variables and the non-basic ones. Strictly speaking, we disallow such internal non-basic variables in S_{b_i} since it implies underspecification (see below).

Proposition 3. $S_b = \mathit{solve}(\Theta_b, \Upsilon_b, \Gamma_b \cup \bigcup_{i=1}^{N} \rho_i(\hat{S}_{b_i})) = \bot$ *iff the set of constraints* $\Gamma_b \wedge \bigwedge_{i=1}^{N} \exists \Upsilon_{b_i}.\rho_i(\Gamma_{b_i})$ *is unsatisfiable.*

If we replace Π in Proposition 1 by $\Gamma_b \cup \bigcup_{i=1}^{N} \rho_i(\hat{S}_{b_i})$, we know that the latter set of constraints is satisfiable iff $\mathit{solve}(\Theta_b, \Upsilon_b, \Gamma_b \cup \bigcup_{i=1}^{N} \rho_i(\hat{S}_{b_i})) \neq \bot$. If $\Gamma_b \wedge \bigwedge_{i=1}^{N} \exists \Upsilon_{b_i}.\rho_i(\Gamma_{b_i})$ is satisfiable, then so is $\Gamma_b \cup \bigcup_{i=1}^{N} \rho_i(\hat{S}_{b_i})$ since the latter is a subset of the former. For the converse, any solution of $\Gamma_b \cup \bigcup_{i=1}^{N} \rho_i(\hat{S}_{b_i})$ yields a solution of $\hat{S}_{b_i}$ by assigning each external variable X in Θ_{b_i} the value of $\rho_i(X)$. By Proposition 2, the latter solution can be recursively extended to a solution for $\Gamma_b \wedge \bigwedge_{i=1}^{N} \exists \Upsilon_{b_i}.\rho_i(\Gamma_{b_i})$.

We say that a subsystem b is underspecified if either one of its component subsystems b_i, for $1 \leq i \leq N$ is underspecified or the assignment of the internal variables of b cannot be uniquely determined from those of Θ_b.

Proposition 4. *If* $S_b = \mathit{solve}(\Theta_b, \Upsilon_b, \Gamma_b \cup \bigcup_{i=1}^{N} \rho_i(\hat{S}_{b_i})) \neq \bot$*, then* $\Gamma_b \cup \bigcup_{i=1}^{N} \rho_i(\hat{S}_{b_i})$ *is underspecified iff* S_b *contains a non-basic internal variable or for some* i*,* $1 \leq i \leq N$*, the component constraint set* S_{b_i} *is underspecified.*

If S_b contains a non-basic internal variable, then by the same argument as the one given for Proposition 2, this variable can be freely assigned any constant dimensional vector value. This assignment is independent of the assignments to the external variables since for each polynomial g_i in S_b where an external variable is basic, i.e., maximal, g_i must only contain external variables. Conversely, if S_b contains no non-basic internal variables, then each basic internal variable Y only occurs in a polynomial of the form $kY + g'$ in S_b, where all the variables in g' are external. Hence, any assignment of values to the external variables is easily extended to an assignment for the basic internal variables.

Finding an Unsatisfiable Core. We can also augment the solver to identify the unsatisfiable core in the case of an inconsistency. This is done by the conventional technique of associating each input constraint with a *zero-slack* variable

ω_i. For example, if the first input constraint is $X + 2Y = 0$ and the second one is $2X + 4Y + \langle L = 0, M = 1, T = 0\rangle = 0$, then these are added with zero slacks ω_1 and ω_2 so that the first equation is rewritten as $X + 2Y + \omega_1 = 0$ and solved as $Y = -(1/2)X - (1/2)\omega_1$. When this solution is substituted into the second equation $2X + 4Y + \langle L = 0, M = 1, T = 0\rangle + \omega_2 = 0$, it yields the normalized form $-2\omega_1 + \omega_2 + \langle L = 0, M = 1, T = 0\rangle = 0$. Since zero-slack variables are never basic variables, this indicates that the input equations 1 and 2 projected on the d dimensions form an unsatisfiable core. In fact, if the inconsistent equation is of the form $k_1\omega_{i_1} + \ldots + k_n\omega_{i_n} + v = 0$, where each k_i and vector v are nonzero, then if we take the corresponding input equations $p_{i_1} = 0, \ldots, p_{i_n} = 0$, we get that $k_{i_1}p_{i_1} + \ldots + k_{i_n}p_{i_n} + v = 0$, yielding the contradiction $v = 0$. The unsatisfiable core given by the set of zero slacks is minimal: if one of the input equations, say $p_{i_j} = 0$, could be dropped from the unsatisfiable core, this means that the M entry of ω_{i_j} can be given any nonzero value. In particular, if ω_{i_j} is assigned $-v/k_j$, that is, $\langle -v_1/k_j, \ldots - v_d/k_j\rangle$ where $v = \langle v_1, \ldots, v_d\rangle$ for d base dimensions, then the equation $k_1\omega_{i_1} + \ldots + k_n\omega_{i_n} + v = 0$ is consistent. Furthermore, this assignment can then be extended to solution set by assigning 0 to all the non-basic variables other than ω_{i_j}. Hence, the constraint $p_{i_j} = 0$ is necessary.

Proposition 5. *If* $S_b = solve(\Theta_b, \Upsilon_b, \Gamma_b \cup \bigcup_{i=1}^{N} \rho_i(\hat{S}_{b_i})) = \bot$, *and the inconsistency arises when an input constraint* $p_i = 0$ *is normalized as* $k_{i_1}\omega_{i_1} + \ldots + k_{i_j}\omega_{i_j} + v = 0$, *where* v *is a non-zero dimension vector and* $k_{i_1}, \ldots, k_{i_j}$ *are non-zero rational coefficients, then normalizing* $k_{i_1}p_{i_1} + \ldots + k_{i_j}p_{i_j}$ *yields* $-v$. *Thus,* $\{p_{i_1} = 0, \ldots, p_{i_j} = 0\}$ *forms an unsatisfiable core of input constraints that is minimal.*

4 Experiments

DimSim performs two major operations: constraints generation and constraints solving. To generate constraints from a Simulink model, DimSim parses the model using Simulink's model construction APIs provided by Mathworks, for example, *find_system* and *get_param*. Using these APIs, DimSim generates constraints sets which are solved using a solver written in Common LISP. The source code for DimSim and the examples are available at `http://www.csl.sri.com/~shankar/dimsim.tgz`.

Categories of Errors. Through dimension analysis, DimSim can discover errors of the following categories:

Erroneous Annotation The user provides the annotations for the interface variables, and erroneous annotations can lead to dimensional inconsistency.

Erroneous Design A wrong design can be detected through dimensional analysis. For example, in the implementation of an *if-then-else* block, the value returned from the *then* block and *else* block should have the same dimension, otherwise the system has some design error. Design errors include missing blocks (e.g., a missing integrator) or an extraneous block.

Model	Model Size				Time Cost		Mismatches found
	No. of Subsystems	No. of Blocks	No. of Variables	No. of Constraints	Constraint Generation	Constraint Solving	
TMH	3	48	79	126	0.663s	0.045s	0
CD2D	9	93	164	212	1.285s	0.142s	1
CC	6	74	139	197	1.028s	0.321s	0
RC	10	102	201	346	1.463s	0.376s	0
ETC	12	113	220	362	1.848s	0.369s	0
TC	34	930	1935	4234	13.984s	11.716s	1

Table 2. Model size, time cost and mismatches found

Incorrect Blocks Usage A Simulink model may contain a wrong block in place of a correct block. This may happen due to adding a wrong block from the library, for example, a Product block is used where a Sum block is required. It may also be due to an incorrect selection of parameters, for example, the Product block may be used for both multiplication and division operations, but to use it for division, the proper sign should be provided for the denominator parameters.

We illustrate the usefulness of DimSim on the following examples: Thermal Model of a House (TMH) [17], Collision Detection System (CD2D) [6], Cruise Control System (CC) [3], Rotating Clutch System (RC) [17], Engine Timing Control System (ETC) [17] and Transmission Control System (TC) [4]. Table 2 summarizes the size of the models,the amount of time taken by different components of DimSim to solve different subproblems related to dimension analysis, and number of dimensional mismatches found in the model. For each model, we report the number of subsystems, number of blocks, number of port variables and number of constraints generated to indicate the size of the model. To compute the time costs, we carried out the experiments in a notebook running Mac OS X version 10.6.7, with 2.26 GHz Intel Core 2 Duo processor and 2GB 1067MHz DDR3 memory. The results show that DimSim is capable of handling large Simulink models in reasonable time.

We found dimensional mismatches in two of our example models. In Table 3, we present the mismatches found, their type, number of constraints in the generated unsatisfiable core and the effectiveness of the unsatisfiable core. The effectiveness of the unsatisfiable core w.r.t. the subsystem for which the set of constraints are inconsistent is defined as the ratio of the number of constraints in the unsatisfiable core and the number of constraints generated from the subsystem. The effectiveness of the unsatisfiable core w.r.t. the model is defined in the similar way. The mismatch in the CD2D model is because the two subsystems corresponding to an *if-then-else* block return variables with different dimensions.[3] In TC, the problem arises for the two input ports of an *Add* block with differing dimensions. The reason is a possible omission of a block that could neutralize the difference.

[3] CD2D is a parametric algorithm where these two branches of the *if-then-else* expression correspond to two distinct modes of the system, each of which is dimensionally consistent. We are grateful to Cesar Muñoz for this clarification.

Error	Model	Type of Error	No. of UC Constraints	Effectiveness w.r.t.	
				subsystem Constraints	total constraints
Error1	CD2D	Erroneous Design	5	19.23%	2.36%
Error2	TC	Erroneous Design	11	5.76%	0.26%

Table 3. Error Data

5 Related Work

Computer Science has a very rich literature on dimension analysis, particularly in the context of general-purpose programming languages, for example, Pascal [5], Ada [8], C++ [18], Fortran [12], Java [20, 7], Fortress [1] and C [9]. For functional languages, Wand and O'Keefe [21] add dimensions and dimension variables to the simply-typed lambda calculus and Kennedy designed dimension types [10] for ML-style languages. Dimensional analysis was also undertaken for simulation language gPROMS [15] and spreadsheets [2]. Mainly two approaches have been used for dimension analysis: (1) Modification of the program source code either based on language extension [5, 20, 1], or using existing language features [8, 18, 12], and (2) Enhancement of the type system using dimensional types and application of unification algorithms using Gaussian elimination to infer dimension types [21, 10, 15, 2, 9, 7].

SimCheck [14], a contract type system for Simulink, supports dimensional analysis of Simulink models through the translation of the model to a set of constraints and uses off-the-shelf decision procedure to solve the constraints. The major differences between SimCheck and DimSim are that SimCheck does not support compositional analysis, cannot detect underspecification and always provides solution in concrete values if feasible, cannot produce an unsatisfiable core in case of an inconsistency, and the analysis in SimCheck is based on a fixed set of dimensions.

DimSim uses a bottom-up compositional strategy to deal with large systems. Among the previous works, only Unifi by Hangal and Lam [7] uses an interprocedural analysis. However, the goal of DimSim is different from that of Unifi. DimSim verifies dimensional consistency in one version of a Simulink model based on user-provided annotations, whereas Unify monitors dimensional relationships between program variables as the program evolves.

Kennedy has developed a dimensional analysis for F# inspired by type inference in ML [11]. In his approach, dimensional constraints are expressed in the theory of Abelian Groups where there is an associative and commutative operator $*$ with an inverse operation $\{\}^{-1}$. For example, the dimensional constraint $m * \alpha^2 = s^2 * \beta$ would express a constraint where m is a constant representing metres, s is a constant representing seconds, and α and β are dimensional variables. Units as used in F# correspond to dimensions in our system in the sense that *metre* is just a name for the length dimension. Finding the most general solution to a dimensional constraint is equivalent to finding the most general unifier for a constraint $u = v$ in the theory of Abelian Groups using a form of Gaussian elimination. For example, unifying $\alpha * metre^2$ with *second* yields $\alpha = metre^{-2} \times second$. In our approach, we solve directly using Gauss–Jordan

elimination to obtain $\alpha = \langle L = -2, M = 0, T = 1\rangle$. This leads to a simpler solver and more informative error reporting. Unlike all previous works based on constraints solving using linear algebra, DimSim provides a concrete explanation of an inconsistency in dimensions and helps the user pinpoint the root cause of such inconsistency.

DimSim also takes a slightly different approach to polymorphic dimensional inference. For example, Kennedy provides an analog of let-polymorphism, where in a function of the form $\lambda y.let\ smult = (\lambda x.x * x * y)\ in\ (smult\ l)/(smult\ t)$, with l representing a length, t a time, and β the dimension of the variable y, the function is given the polymorphic type $\forall\alpha.[\alpha{\rightarrow}\alpha^2 * \beta]$. Our analysis can also be adapted to admit this kind of let-polymorphism. For example, the expression $x * x * y$ can be seen as having input variables x and y, and a dummy output variable z. This yields the dimensional constraint $Z = 2X + Y$ so that *smult* has the type $\forall X.[X{\rightarrow}[Z : Z = 2X + Y]]$. The dimension of $(smult\ l)$ is $[Z : Z = 2\langle L = 1, M = 0, T = 0\rangle + Y]$, and that of $(smult\ t)$ is $[Z : Z = 2\langle L = 0, M = 0, T = 1\rangle + Y]$. The quotient $(smult\ l)/(smult\ t)$ can be given the dimension $[W : W = 2\langle L = 1, M = 0, T = -1\rangle]$, and the entire lambda-expression has the dimension $[Y{\rightarrow}[W : W = 2\langle L = 1, M = 0, T = -1\rangle]]$.

6 Conclusions

We have presented DimSim , an automatic dimension analyzer for cyber-physical systems modeled in Simulink. DimSim employs compositional analysis technique to deal with large size Simulink model, and in case of an inconsistency, provides a proof of the inconsistency locally on the offending subsystems. Our case studies on numerous examples show that DimSim does find modeling errors and has the potential to be used in the industrial context.

Our approach to dimensions is pragmatic. We have focused on finding dimension errors in Simulink models of cyber-physical systems. Our dimension system is parametric in the choice of the base dimensions. As an example, in physical terms, angles are treated as dimensionless, but in our system, it is possible to introduce angles as a base dimension. Using angles as a dimension allows certain classes of bugs to be found, but it could be incompatible with calculations of values of trigonometric functions based on their Taylor expansion.

We have not yet extended DimSim to handle units of dimensions such as feet and metres. There are several approaches to incorporating units. One approach (e.g., Kennedy [11]) is to treat each such basic unit as an independent basic dimension so that it is possible to mix different units within a single dimension expression and use arbitrary conversions between units. Another approach is to report an error when different units for the same dimension are mixed. We could also restrict conversions between units to those that are derived using specific scaling and offset operations. We plan to experiment with several different approaches to units in DimSim in order to identify the criteria that works best with cyber-physical models.

References

1. E. Allen, D. Chase, V. Luchangco, J. Maessen, and G. L. Steele, Jr. Object-oriented units of measurement. In *Proceedings of OOPSLA*, pages 384–403, 2004.
2. T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the unit correctness of spreadsheet programs. In *Proceedings of ICSE*, pages 439–448, 2004.
3. K. J. Astrom and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2009.
4. A. Chutinan and K. R. Butts. Smart vehicle baseline report - dynamic analysis of hybrid system models for design validation. Technical report, 2002.
5. A. Dreiheller, B. Mohr, and M. Moerschbacher. PHYSCAL: Programming Pascal with physical units. *SIGPLAN Not.*, 21:114–123, 1986.
6. A. L. Galdino, C. Muñoz, and M. Ayala-Rincón. Formal verification of an optimal air traffic conflict resolution and recovery algorithm. In *Proc. of WoLLIC*, pages 177–188, 2007.
7. S. Hangal and M. S. Lam. Automatic dimension inference and checking for object-oriented programs. In *Proceedings of ICSE*, pages 155–165, 2009.
8. P. N. Hilfinger. An ADA package for dimensional analysis. *ACM Trans. Program. Lang. Syst.*, 10:189–203, 1988.
9. L. Jiang and Z. Su. Osprey: A practical type system for validating dimensional unit correctness of C programs. In *Proceedings of ICSE*, pages 262–271, 2006.
10. A. Kennedy. Dimension types. In *Proceedings of ESOP*, pages 348–362, 1994.
11. A. Kennedy. Types for units-of-measure: Theory and practice. In *CEFP*, pages 268–305, 2009.
12. G. W. Petty. Automated computation and consistency checking of physical dimensions and units in scientific programs. *Softw. Pract. Exper.*, 31:1067–1076, 2001.
13. J. J. Roche. *The Mathematics of Measurement: A Critical History*. Springer, 1998.
14. P. Roy and N. Shankar. SimCheck: A contract type system for Simulink. *ISSE*, 7(2):73–83, 2011.
15. M. Sandberg, D. Persson, and B. Lisper. Automatic dimensional consistency checking for simulation specifications. In *Proceedings of SIMS*, 2003.
16. Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 3rd edition, 2003.
17. The MathWorks. Simulink - Demos. `http://www.mathworks.com/products/simulink/demos.html`.
18. Z. D. Umrigar. Fully static dimensional analysis with C++. *SIGPLAN Not.*, 29:135–139, 1994.
19. L Valiant and V Vazirani. NP is as easy as detecting unique solutions. *Theoretical Computer Science*, pages 85–93, 1986.
20. A. Van Delft. A Java extension with support for dimensions. *Softw. Pract. Exper.*, 29:605–616, 1999.
21. M. Wand and P. O'Keefe. Automatic dimensional inference. In *Computational Logic-Essays in Honor of Alan Robinson*, pages 479–483, 1991.