A Compositional Modeling Language

Daniel Bobrow[*], Brian Falkenhainer[**], Adam Farquhar[#], Richard Fikes[#], Kenneth Forbus[$], Thomas Gruber[&], Yumi Iwasaki[#], Benjamin Kuipers[%]

[*]Xerox Corporation Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
bobrow@parc.xerox.com

[**]Xerox Wilson Center
800 Philips Rd., M/S 128-51E
Webster, NY 14580
falken@wrc.xerox.com

[#]Knowledge Systems Laboratory
Gates Bldg. 2A, M/C 9020
Department of Computer Science
Stanford University
Stanford, CA 94305
{axf, fikes, iwasaki}@ksl.stanford.edu

[$]Qualitative Reasoning Group
The Institute for the Learning Sciences
Northwestern University
1890 Maple Avenue
Evanston, IL 60201
forbus@ils.nwu.edu

[&]Colloquy Systems Inc.
5150 El Camino Real, Suite D-21
Los Altos, CA 94022
gruber@colloquy.com

[%]University of Texas at Austin
Department of Computer Science
Austin, TX 78712
kuipers@cs.utexas.edu

Abstract

This document describes a compositional modeling language, CML, which is a general declarative modeling language for logically specifying the symbolic and mathematical properties of the structure and behavior of physical systems. CML is intended to facilitate model sharing between research groups, many of which have long been using similar languages. These languages are based primarily on the language originally defined by Qualitative Process theory [Forbus 1984] and include the languages used for the Qualitative Physics Compiler (QPC) [Crawford 1990; Farquhar 1993; Farquhar 1994], compositional model formulation [Falkenhainer 1991], and the Device Modeling Environment (DME) [Low and Iwasaki 1993]. CML is an attempt to synthesize and provide a clean redesign of these languages.

1. Introduction

Compositional modeling is an effective paradigm for formulating a behavior model of physical system by composing descriptions of symbolic and mathematical properties of individual system components. This paper describes Compositional Modeling Language (CML), which is a general declarative modeling language for representing physical knowledge required for compositional modeling.

CML is intended to facilitate model sharing between research groups, many of which has long been using similar languages. These languages are based primarily on the language originally defined by Qualitative Process Theory [Forbus 1984] and include the languages used for the Qualitative Physics Compiler [Farquhar 1994], compositional model formulation [Falkenhainer 1991], and the Device Modeling Environment [Low and Iwasaki 1993]. CML is an attempt to synthesize and provide a clean redesign of these languages. The specification of CML has been formulated by researchers involved in those projects.

CML was designed with efficiency, expressiveness and ease of use in mind. The language is restricted enough to allow efficient implementation of procedures to predict behavior. The syntax is simple and readable so that a person familiar with the domain will be able to read and easily understand an expression of knowledge of the domain in the language. The language supports lumped parameter ordinary differential equations that are common in engineering modeling. Finally, the language supports a variety of different approaches to representing physical phenomena; it allows the definition and use of domain theories that use components, process, bond graphs, kinematic pairs, etc., and also supports both relational and object-oriented specification styles.

CML specifies a set of top-level forms for defining models and an ontology of primitive functions, relations, and constants. CML is intended to be an open, evolving language, of which this document describes the *base language*. Various extensions will undoubtedly be defined

as they naturally arise in the course of its use by different people. An important goal in designing the base language is to support as much sharing as is reasonably possible. Also, to facilitate sharing the content of CML knowledge bases, CML is fully translatable to the knowledge interchange format[1] (KIF)[Genesereth and Fikes 1992], and we have adopted conventions established by KIF wherever possible.

1.1. Patterns of Use

A typical implementation supporting CML might be used as follows: To predict the behavior of a physical system in some domain, knowledge about the physics of the domain is captured in a general purpose *domain theory* that describes classes of relevant objects, phenomena and systems. The domain theory of chemical processing plants, for example, might include physical phenomena such as mass and heat flows, boiling, evaporation, and condensation; it would also include chemical reactions, the effects of catalysts, and models of components such as reaction vessels, pumps, controllers, and filters. A domain theory in CML consists of a set of quantified definitions, called *model fragments*, each of which describes some partial piece of the domain's physics, such as processes (e.g., liquid flows), devices (e.g., transistors), and objects (e.g., containers). Each definition applies whenever there exists a set of participants for whom the stated conditions are satisfied. A specific system or situation being modeled is called a *scenario*. A model of the scenario consists of fragments that logically follow from the domain theory and the scenario definition.

For example, consider the situation depicted in Figure 1. A scenario representing this situation would state that there is a can containing some water placed over a gas heater. In addition, the scenario may also state whether or not the gas heater is initially on, the initial temperature and volume of the water and so on. In order to reason about this situation, the domain theory must contain the definitions of a can, contained water, a gas heater, as well as the definitions of relevant physical processes such as heat flow and evaporation. The definitions of these objects and processes must specify their numeric and non-numeric attributes, such as *water-level* and *flame-lit-p*. The types of values such attributes take, for example "a numeric, time-dependent quantity whose dimension is length" must also be specified in the domain theory.

Once the domain theory has been constructed, it can be used to model many different physical devices under a variety of different conditions. The user specifies a scenario that defines an initial configuration of the device, the initial values of some of the parameters that are relevant to modeling it, and perhaps conditions that further characterize the system. The CML implementation would automatically identify model fragments that are applicable in the scenario. These model fragments would be composed into a single model that comprises both a symbolic description as well as a set of governing equations. The equations may be solved or simulated to produce a behavioral description. Because the conditions under which the model fragments hold are explicit in the domain theory, the system would be able to construct automatically additional models that describe the device as it moves into new operating regions.

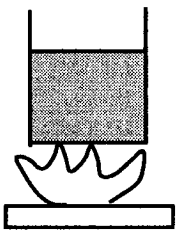

Figure 1: An example situation with a can of water and a heater

1.2. Notation and Syntax

The CML syntax is based on the Common Lisp standard [Steele 1990]; a sequence of characters is a legal CML expression only if it is acceptable to the Common Lisp reader with standard settings. In this document, we will adopt the following notational conventions: Variables are marked with a ? prefix, to distinguish them from object and relation constants. Where the syntax allows for a finite series of items indexed from *1* to *n*, the first item of the sequence is given with the subscript *1* and the remaining *n-1* items are abbreviated by "...n". For example,

((*participant*$_1$:type *type*$_1$) ...n)

is the notation for

((participant$_1$:type type$_1$) ...

(participant$_n$:type type$_n$)).

2. General Semantics

The goal of CML is to provide a common syntax with a well-defined semantics so that different implementations, with different internal representations and inference procedures, will be able to accurately parse the same domain theory. The syntax and semantics of CML top-level forms is presented in Section 3. The semantics of a set of CML top -level forms is provided by translating them into first order logic such as defined in KIF, and CML inherits the logic's model-theoretic semantics. In this section, we describe the general semantics of relations, individuals and quantities underlying the definitions of the top-level forms.

[1] KIF provides a standard encoding and semantics for a first order logic with set theory and some minor extensions such as a restricted quote and the ability to refer to relations directly.

2.1. Quantities

CML is designed to model time-varying physical systems, such as the movement of a mechanical device or the process of a chemical reaction. In engineering models, the properties and state of such systems are described by variables, parameters, coefficients, and constants. In CML, the term *quantity* encompasses these notions. A quantity is either a constant or a unary function whose argument is a time.

In the syntax of CML, time is left as an implicit parameter to time-dependent quantities, functions, and relations. All non-constant quantities are restricted to have a finite number of critical points or discontinuous changes over any finite interval. This restriction rules out certain classes of poorly behaved systems, such as oscillators with infinite frequency, that pose problems for numeric integration and qualitative simulation techniques.

Quantities may be numeric or non-numeric. *Non-numeric quantities* are simply constants or unary functions of time satisfying the above finite-change requirement. Their values are unrestricted. A *numeric quantity* is associated with a single physical dimension, given by the function `dimension`.

CML specifies a core set of fundamental physical dimensions: the seven defined by the System Internationale (mass, length, time, charge, temperature, amount, and luminosity) plus a dimension for dimensionless numbers. Real numbers are dimensionless constant quantities. CML also provides a top-level form to allow definition of any other dimensions. A dimension is a property that is used to distinguish incompatible quantities. Quantities of the same dimension can be compared, added, and so on. These operations are not defined for quantities of different dimensions.

A mathematical relation holds on non-constant quantities if it holds on their values at each time that they are defined. A *numeric time-dependent quantity* is a function of time whose values all have the same dimension. The value of a numeric time-dependent quantity is a numeric constant quantity.

The magnitude of a *numeric constant quantity* is specified in units of measure. A *unit of measure* is itself a constant quantity used as a reference for a given dimension. For example, the meter is a unit of measure for the length dimension and the second is a unit of measure for the time dimension. The magnitude of a constant quantity depends on the unit in which it is requested. The binary function *magnitude* maps a constant quantity and a unit of the same dimension to a real number. For example, the magnitude of *12g* in grams is *12* and its magnitude in ounces is about *4.23*. A unit of measure defines an absolute scale with a *0* value for quantities of a particular dimension. The real number *0* is dimensionless and therefore is different from other quantities whose magnitude is *0*, such as *0 Newtons* or *0 feet*.

CML supports *everywhere continuous quantities, piecewise continuous quantities, step quantities*[1], and *count quantities*[2].

2.1.1. Handling implicit dependence on time

One important aspect of translating the semantics of quantities into logic is the representation of time. A *time-quantity* is a numeric, everywhere continuous quantity whose dimension is the *time-dimension.* All time-dependent quantities and relations in CML have a time-quantity as an implicit argument. In the translation of CML into logic, time is handled in three steps:

1. Every time-dependent relation is augmented with a first argument, which must be a time-quantity.
2. Every time-dependent quantity, q, is uniformly translated with the following form, where *value-at* is a function of two arguments, a quantity and a time, and returns the value of the quantity at the time:

   (lambda (?t1) (if (= ?t1 ?t) (value-at q ?t1)).
3. Every mathematical function that typically applies to numbers (constant quantities) is polymorphically extended to apply to function quantities as well. Every mathematical operation that typically applies to numbers must be extended similarly[3].

3. Language Definition

A domain theory in CML is a finite *set*[4] of the following top-level forms:

defRelation for defining logical relations.

defQuantityFunction for defining quantities used in the domain theory.

defModelFragment for describing the behavior of modeled entities under explicitly specified conditions. Model fragments are used to describe phenomena that arise out of the interactions of a composite set of objects (e.g., collisions or flows), or the behavior of a single object (e.g., a resistor, pump, or valve).

defEntity for defining properties of persistent objects (e.g., resistors, containers).

defScenario for defining initial value problems consisting of a set of objects, their configuration, and initial values for the quantities that describe them.

In addition, the forms **defDimension**, **defUnit**, and **defConstantQuantity** are provided for defining new or

---

[1] Step quantities are piecewise continuous quantities that are constant over every continuous interval.

[2] Count quantities are step quantities that have non-negative integer values and are dimensionless.

[3] Note that such a mathematical relation is false when any of its arguments is undefined.

[4] Thus, implementations must allow for use before definition.

derived dimensions, new or derived units, and universal constants, respectively.

The general syntax of a form is the form identifier (e.g., **defQuantityFunction**), followed by its name, followed by a series of keyword/value pairs. Some keywords are optional, as indicated by the surrounding brackets in the grammar (i.e., [**:dimension**]).

Wherever a keyword/value pair appears, an arbitrary number of other, implementation specific, keywords are allowed. If the domain theories employing such keywords are to be portable, however, the following restriction must be satisfied. If the keywords affect the behavioral inferences entailed by the domain theory, then they should only *strengthen or annotate* the *behavioral inferences*. This allows other implementations to ignore the additional keywords and still draw *correct*, if weaker conclusions. Domain theories that employ an extension that satisfies this criterion will be sharable across implementations.

The rationale is that various groups will want to add fields that facilitate explanation tools or support software engineering of large knowledge bases (e.g., short names, authors, pointers to extended documentation, etc.). A secondary purpose is to allow for local extensions to the language as the need arises, without having to change the common language specification. The impact such extensions may have on sharability and the semantics of a given domain theory is not addressed here. In some cases, particularly qualitative simulation, the added information may simply provide monotonic reductions in ambiguity. In other cases, particularly in the presence of a closed world assumption, the nature of the predicted behavior may be fundamentally affected.

Table 1 shows the syntax for the top-level forms with some examples. The definitions for the intermediate forms used in this table are given in Table 2. The following sections discuss each form in detail.

3.1 Relations and Functions

In the **defRelation** form, the symbol *Name* is a global relation constant naming a relation of arity n; the x_i are logical variables, one for each argument of *Name*. All uses of *Name* in the domain theory should be consistent with the specified constraints.

- => The *asentence* is a logical sentence that may mention the variables x_i. If present, the sentence (=> (*Name* $x_i \ldots x_n$) *asentence*) is true.
- <=> The *asentence* is a logical sentence that may mention the variables x_i. If present, the sentence (<=> (*Name* $x_i \ldots x_n$) *asentence*) is true.
- **Function** If **function** is **true**, then the relation *Name* is a function. That is, the first *n-1* arguments to *Name* uniquely determines the nth and (*Name* $x_1 \ldots x_{n-1}$) may be used as a term denoting x_n.
- **time-dependent** If **time-dependent** is **true**, then *Name* is a time-dependent relation and appearances of *Name* in a CML form must be handled specially.

3.2. Quantity Functions

The **defQuantityFunction** form defines a function that maps a tuple of objects to a quantity. The quantity is itself a function of time. In addition to being globally defined via **defQuantityFunction**, quantity functions may also be defined within model fragment and entity definitions in the **quantities** clause.

The *name* is a function constant naming an n-ary function that returns a time-dependent quantity. The $x_1 \ldots x_n$ are logical variables.

- => The *asentence* is a logical sentence that may mention the variables x_i. It may be used to place restrictions on the quantity's values, or assert things, such as type information, about the x_i.
- **Dimension** The *dimension expression* (see **defDimension** for the complete syntax) specifies the dimension of the quantities returned by the quantity function.
- **Non-numeric** If **non-numeric** is **true**, then the quantities returned by the function are non-numeric, otherwise they are numeric.
- **Piecewise-continuous** If **piecewise-continuous** is **true**, then the quantities returned by the function are piecewise-continuous.
- **Step-quantity** If **step-quantity** is **true**, then the quantities returned by the function are step-quantities.
- **Count-quantity** If **count-quantity** is **true**, then the quantities returned by the function are dimensionless count-quantities.

3.3. Model Fragments and Entities

This section defines the syntax and semantics for the key CML forms **defModelFragment** and its restricted version **defEntity**.

The **defModelFragment** form defines a class of phenomena, which are described by a set of objects involved, static attributes and time-dependent quantities. It also defines consequences that hold only when an instance of the class is *active*. The **defModelFragment** form may define conditions sufficient to imply the existence of an instance, in addition to the necessary consequences thereof.

The **defEntity** form is a restricted version of **defModelFragment** that is used for defining properties of a persistent object that are *always true*. The **defEntity** form defines only necessary consequences of an object being an instance of the class, not conditions sufficient to imply the existence of an instance.

The **defEntity** and **defModelFragment** forms have been designed to support an object-oriented style of defining domain theories. Each form defines a class of objects specified by sets of static **attributes** and time dependent **quantities**. These attributes and quantities are effectively slots defined on instances of the class. Furthermore, these classes may be arranged in a hierarchy via the **subclass-of** clause.

Syntax	Examples
(defRelation *Name* (x_1 ... x_n) [**:documentation** *string*] [**:=>** *asentence*] [**:<=>** *asentence*] [**:function** *boolean*] [**:time-dependent** *boolean*])	(defRelation contents (?x ?y) :=> (and (container ?x) (contained-stuff ?y)) :<=> (contained-in ?y ?x)) (defRelation fahrenheit (?t ?f) :<=> (== ?f (- (magnitude ?t rankine) 459.7)) :function true) (defRelation above (?x ?y) :time-dependent true)
(defQuantityFunction *Name* (x_1 ... x_n) [**:documentation** *string*] [**:=>** *asentence*] [**:dimension** *dimension expression*] [**:piecewise-continuous** *boolean*] [**:step-quantity** *boolean*] [**:count-quantity** *boolean*] [**:non-numeric** *boolean*])	(defQuantityFunction mass (?x) :=> (physical-object ?x) :dimension mass-dimension) (defQuantityFunction density (?x) :=> (physical-object ?x) :dimension (/ mass-dimension (expt length-dimension 3)))
(defModelFragment *Name* [**:documentation** *string*] [**:subclass-of** *class*$_1$ ...s] [**:participants** ((*participant*$_1$ [**:type** *type*$_1$])...p)] [**:conditions** *conditions*] [**:quantities** ((*quantity*$_1$ *keywords* $_1$) ...q)] [**:attributes** ((*attribute*$_1$ [**:type** *attr type*$_1$])...a)] [**:consequences** *consequences*])	(defModelFragment Contained-Stuff :subclass-of (physical-object) :participants ((sub :type substance) (ctnr :type fluid-container)) :conditions ((> (amount-of-in sub ctnr) (* 0 grams))) :quantities ((pressure :dimension pressure-dimension) (mass :dimension mass-dimension)) :consequences ((== mass (amount-of-in sub ctnr))))
(defEntity *Name* [**:documentation** *string*] [**:subclass-of** *class* $_1$ ...s] [**:quantities** ((*quantity*$_1$ *keywords* $_1$) ...q)] [**:attributes** ((*attribute*$_1$ [**:type** *attr type*$_1$])...a)] [**:consequences** *consequences*])	(defEntity Can :subclass-of (physical-object Container) :quantities ((height :dimension length-dimension) (diameter :dimension length-dimension) (volume :dimension volume-dimension)) :consequences ((== (volume :self) (* PI (expt (/ diameter 2) 2) height))))

Table 1: Syntax and examples of top-level forms

Syntax	Examples
(**defDimension** *Name* [**:documentation** *string*] [:= *dimension expression*]) (**defUnit** *Name* [**:documentation** *string*] [:= *quantity expression*] [**:dimension** *dimension expression*]) (**defConstantQuantity** *Name* [**:documentation** *string*] [:= *quantity expression*] [**:dimension** *dimension expression*])	(defDimension energy-dimension := (* mass-dimension length-dimension (expt time-dimension -2))) (defUnit inch := (* 2.54 (* meter 0.01))) (defConstantQuantity Pi := (acos -1)) (defConstantQuantity Boltzman-Constant := (* 1.380658 (/ Joule Kelvin)))
(**defScenario** *Name* [**:documentation** *string*] [**:individuals** ((*individual* [**:type** *type*])*)] [**:initially** *asentences*] [**:throughout** *asentences*]	(defScenario water-heating-example :documentation "A can containing water is placed directly above a gas-heater, which is initially lit." :Individuals ((C :type Can) (H :type Gas-heater) (W :type Contained-water)) :initially ((> (amount-of-in W C) (* 0 grams))) :throughout ((= (height A) (* 0.2 meter)) (= (diameter A) (* 0.15 meter)) (flame-lit-p H true) (directly-above C H)))

Table 1: cont.

Additional Syntax

dimension expression ::=
- *dimension* |
- (* *dimension expression* +) |
- (expt *dimension expression* *number*) |
- (/ *dimension expression* *dimension expression*)

quantity expression ::=
- *unit* | *quantity* |
- (*mathop* *quantity expression* +)

asentence ::= (and *literal**)

literal ::=
- (*relconst* *term**) |
- (not (*relconst* *term* *))

Table 2: Some Additional Syntax

Name The name of the model fragment or entity, *name*, is a relation constant naming the class of instances.

Subclass-of The **subclass-of** clause allows a hierarchy to be defined. Each *class* is the name of a model fragment or entity definition. An instance of *name* is also an instance of each superclass. As a consequence, all of the participant, quantity, and attribute functions defined for each *super* are also defined for *name*.

Participants The **participants** clause identifies the objects that participate in the model fragment instance. Each **participant** is a function constant that names a unary function which may be applied to an instance to access the corresponding participant; each *type* is a relation constant that names a class (unary relation) of which the participant is an instance.

Conditions The **conditions** clause specifies the conditions under which an instance of a model fragment is active. If the conditions hold over the specified participants, then an instance of the model fragment exists with the specified quantities and attributes. *Conditions* is an implicit conjunction of literals. The binary relations **same** and **different** may be used in the **conditions** to state that two participants are the same or different from each other.

Attributes The **attributes** clause may be used to define static attributes of an instance. Each *attribute* is a symbol naming a function that is totally defined for instances of *name*. The attributes are polymorphic, that is, an attribute of the same name may be defined for another unrelated form with a different type.

Quantities The **quantities** clause may be used to locally define quantities that describe an instance. The *QF keywords* are the keyword options defined for **defQuantityFunction**, except that => is not allowed. Such implications may be placed in the **consequences** clause. The quantities are polymorphic, that is, a quantity of the same name may be defined for another unrelated model fragment, but have different properties. Nonetheless, quantity functions defined in a **quantities** clause must be consistent with any of the constraints imposed by a **defQuantityFunction** definition of the same name. For example, entity definitions Liquid and Sand can both define a quantity called Amount-of with dimensions volume-dimension and mass-dimension, respectively. However, if there is a separate global definition of Amount-of using the **defQuantityFunction** form, which specifies the dimension to be volume-dimension, the quantity definition of Amount-of in Sand is disallowed since it is inconsistent with the global definition

Consequences The **consequences** clause holds whenever an instance is active. The *consequences* is an implicit conjunction of literals. The primary role of the consequences is to establish equations that help to define the behavior of the participants. In addition to equations, other logical relations may also be asserted.

3.3.1 Syntactic Sugar

In order to allow for more concise and readable definitions, the **defEntity** and **defModelFragment** forms provide some syntactic sugar.

Self The symbol **self** may be used to refer to the current instance. Note that it may not be used in the **conditions** clause of a model-fragment definition with no superclasses, as this would place it outside of the scope within which the instance exists.

Name The user provided symbol for the *name* of a model fragment or entity may be used instead of **self** and is completely equivalent.

Quantity The symbol for any *quantity* may be used to refer to the appropriate quantity within the **consequences** clause. This is completely equivalent to the more verbose form (*quantity* `self`), which may also be used.

Attribute The symbol for any *attribute* may be used to refer to the appropriate attribute within the *consequences* clause. This is completely equivalent to the more verbose form (*attribute* `self`), which may also be used.

Participant In a model fragment definition, the user provided symbol for each *participant* may be used to refer to that *participant*. Outside of the *conditions*, the more verbose form (*participant* `self`) may also be used.

3.4. Semantics

The full semantics of the CML forms are defined in [Falkenhainer, Farquhar et al. 1994]. We provide an informal account of them here, starting with the simpler defEntity form.

The **defEntity** form defines a class of objects. If any object is a member of the class, then the quantities and attributes defined in the form apply to it, and the consequences are true for them. Entities may be structured into a hierarchy using the **subclass-of** clause; all quantities, attributes, and consequences that apply to a superclass also apply to the subclass. That is, all inheritance is monotonic — there is no way to over-ride default values that are inherited.

A **defModelFragment** form without any superclasses is also simple to understand. If the participants exist and satisfy the time-independent conditions, then an instance of the model fragment exists. At any moment that the time-dependent conditions hold, the model instance is active and the consequences hold. If the time-dependent conditions do not hold, the consequences are not implied. A **defModelFragment** form without superclasses defines sufficient conditions for an instance to exist.

A **defModelFragment** form with superclasses is somewhat more complex. If there is some object that is an instance of all of the definition's superclasses and the definition's participants exist and its time-independent conditions hold, then that object is *also* an instance of the definition. Activity and consequences are handled just as for model fragments without superclasses. A **defModelFragment** form with superclasses defines necessary conditions for an object to be an instance.

The previous paragraphs describe CML as it has been defined and is consistent with its predecessor languages. This scheme is extremely useful for providing additional information about concrete physical phenomena in a library. For instance, a library might include one definition for fluid-flow that held whenever there were two containers connected by a port. Subclasses of fluid-flow might include laminar flow, turbulent flow, and so on.

This scheme, however, has an important shortcoming that it does not allow abstract model fragments to be defined. To understand this difficulty, consider an example of a library of chemical reactions. Such a library might include model fragments for binary chemical reactions, such as oxidation, between substances. There are a few things that can be said about all binary chemical reactions such as "there are two distinct reactants". Thus, Binary-reaction, the class of all binary reactions, may have two participants, Reactant-1 and Reactant-2 and a condition that they are distinct. It is not natural, however, to specify furthere conditions under which a generic binary reaction occurs. This is much easier to say about a specific chemical reaction. Oxidation, a subclass of Binary-reaction, may have the condition that Reactant-1 is an oxidant, and Reactant-1 and Reactant-2 are in contact. Given a situation involving three chemical substances, A,

B, and C, such that only A is an oxidant, and A and B are in contact with each other, one would expect existance of only one binary chemical reaction, which is also an oxidation reaction, to be inferred. However, from the semantics of the model fragments described above, there would be six instances of abstract binary reactions, one for each possible combinarion of A, B, and C. Although the current interpretation is coherent and logically consistent, it poses a practical problem that it enables a large number of uninteresting model fragment instances to be inferred. We are currently considering an alternate scheme that supports abstract model fragments with or without superclasses.

3.5. Dimensions, Units, and Constants

The vocabulary used to describe quantities varies from one domain to another. For this reason, it is essential to be able to define new dimensions and units. Often, these will be derived from the base set of SI dimensions and units (e.g., an electro-magnetic domain theory might define a dimension for magnetic-flux and its SI derived unit, the Weber). If the new dimension is reducible to other dimensions, the dimension expression must be provided. The top-level forms **defDimension** and **defUnit** provide this facility. The form **defConstantQuantity** is also provided for defining global named constants.

Every CML implementation should have a built-in library of definitions for the basic SI dimensions and units: **time-dimension**, **length-dimension**, **temperature-dimension**, **mass-dimension**, **luminosity-dimension**, **charge-dimension**, **amount-dimension** (usually measured in moles), and **dimensionless**. The library should also include the definitions for the common SI units including the base units, **meter**, **kilogram**, **second**, **ampere**, **Kelvin**, **mole**, as well as the derived units **Hertz**, **Newton**, **Pascal**, **Joule**, **Watt**, **Coulomb**, **volt**, **Farad**, **ohm**, **Siemens**, **Weber**, **Tesla**, and **Henry**.

Except for **dimensionless**, dimensions are, by convention, named by affixing **-dimension** to the standard English word. This makes it straightforward to distinguish between dimensions and similarly named quantity functions.

If a **defUnit** lacks the = argument, then it defines a fundamental unit. A fundamental unit definition must have either a **dimension** (as in the **meter** example) or a *quantity expression*, in which case the dimension is inferred from that of the *quantity expression*. If the expression is complex, it may be more informative to provide the dimension explicitly.

The **defConstantQuantity** form is identical to **defUnit**, except that = must be provided.

3.6. Scenarios

The **defScenario** form is used for setting up problems in which the behavior of a system is to be predicted from a set of initial conditions.

Individuals The **individuals** clause specifies a set of named objects that are assumed to exist initially. The domain theory may imply the existence of other individuals. The *individual* is an object constant denoting an object, *not a relation or function*.

Initially The **initially** clause specifies conditions that initially hold in the scenario. It is an implicit conjunction of literals. It may specify relations between quantities, time-dependent relations, and perhaps an assignment for the quantity **time**.

Throughout The **throughout** clause specifies conditions that hold throughout the scenario. It is a list of literals under an implicit conjunction.

4. Equations

CML provides a base set of mathematical functions and operators that can be used in the consequences and conditions of model fragment and entity definitions as well as initially and throughout clauses of scenario definitions. The mathematical functions and relations can be applied to time dependent variables as well as time independent ones. The relations on quantities included: <, <=, >=, >, ==, positive, negative, zero, integer, odd, even. CML includes mathematical functions as defined in the Common Lisp standard (+, -, *, /, abs, acos, acosh, asin, asinh, atan, atanh, cos, cosh, exp, expt, log, max, min, mod, signum, sin, sinh, sqrt, tan, tanh); the time derivative d/dt; the qualitative relations M+, M- M+0, M-0, Q=; and the composable equations C+, C-, Qprop+, Qprop+0, Qprop-0, I+, I. The full semantics of these functions and relations are defined in [Falkenhainer, Farquhar et al. 1994]. Table 3 summarizes the derivative, qualitative constraints, and composable equations.

5. Conclusion

In this paper, we have presented CML, the knowledge representation language for the compositional modeling paradigm. The main advantage of compositional modeling is its modularity. Writing model fragments, each describing a single phenomenon, is a much easier task than composing a complete model for every possible system and query. Even so, constructing such a domain theory is a substantial undertaking. Thus, the major goal of CML is to support the interchange and reuse of such theories. To enable sharing of knowledge stated in CML, the semantics of CML is fully defined in KIF.

The language presented here is the base language of CML, which all the implementations of modeling systems using CML are expected to support. Various extensions to the base language will undoubtedly be needed to accommodate domain-specific representational needs. Some of such extensions that we are considering are the following:

Forms	Examples	Meaning
(d/dt x) (d/dt x y)	(> (d/dt (location m) 0) (d/dt (pos m) (vel m))	The time derivative of x. The time derivative of x is y.
(M+ y x) (M- y x)	(M+ (level w) (pressure w)) (M- (vol w) (space c))	The quantity y is a in(de)creasing monotonic function of x.
(M+0 y x) (M-0 y x)	(M+0 (level w) (pressure w))	Y is a monotonic in(de)creasing function of x, but passes through the origin.
(Q= x y)		X and y and their derivatives have the same sign. A somewhat weaker statement than M+0.
(C+ y x) (C- y x)	(C+ (net c) (in 1 c)) (C- (net c) (out c))	Composable addition and subtraction. X in(de)crements y. Note that these can be mixed with Qprops.
(Qprop+ y x) (Qprop- y x)	(Qprop+ (size (drain c)) (out c))	Composable qualitative proportionalities. Note that these can be mixed with C+, C-. Qprop+ is equivalent to a C+ chained to an M+.
(Qprop+0 y x) (Qprop-0 y x)	(Qprop+0 (current w) (conductance w))	Similar to Qprop, but if all of the composed equations on y are C+, C, Qprop+0, Qprop-0, then y=0 when the x's are 0.
(C* y x) (C/ y x)	(C* (magnification scope) (magnification lens))	Similar to C+, C-, but x multiplies (divides) into y.
(correspondence y v x_1 v_1 ... x_n v_n)	(correspondence (level c) top (volume c) full)	If there is a function f such that $y=f(x_1 \ldots x_n)$ then when $x_i=v_i$, y=v. The x_i are quantity functions, and the v_i are values.

Table 3: Qualitative Relations and Compositional Equations

- Extension to the representation of quantities to include vectors and matrices.
- Extension to the notion of mathematical functions to include mathematical relations represented by arbitrary external data structures such as databases and foreign procedures.
- Expansion of the mathematical vocabulary to include partial differential equations.
- Generalization of the model fragment representation to represent non-numeric and discontinuous changes.
- Representation of procedural specifications such as prescribed operator procedures.

The detailed specification of the syntax and semantics of the language along with discussions of design rationale and implementation considerations can be found in [Falkenhainer, Farquhar et al. 1994]. To facilitate construction of domain theories in CML, we have implemented a web-based CML editor for browsing, creating and editing CML domain theories. The editor, which is publicly accessible on the WWW, provides a full, distributed collaborative editing environment. We are also in the process of implementing a model formulation and simulation system, which will make use of the CML library. The system will be also publicly accessible as a service on the WWW. We hope that the availability of these services will facilitate development of a significant public library of domain theories by this research community and that it will spur further research and development in the field.

Acknowledgement

The authors thank Vijay Saraswat for useful comments on the language specifications. The research by the authors are supported in part by the following agencies: Forbus by the Office of Naval Research, Farquhar, FIkes, Gruber, and Iwasaki by ARPA and NASA/ARC under contract NAG2-581 (ARPA order 8607), and Kuipers by NSF grants IRI-9216584 and IRI-9504138 and by NASA grants NCC 2-760 and NAG 2-994.

References

Crawford, J., Farquhar, A., and Kuipers, B. (1990). QPC: A Compiler from Physical Models into Qualitative Differential Equations. The Eighth National Conference on Artificial Intelligence,

Falkenhainer, B., Forbus, K. (1991). "Compositional modeling: finding the right model for the job." Artificial Intelligence **51**(1-3):

Falkenhainer, B., Farquhar, A., Bobrow, D., Fikes, R., Forbus, K., et al. (1994). CML: A Compositional

Modeling Language. Technical report KSL-94-16, Knowledge Systems Laboratory, Stanford University.

Farquhar, A. (1993). Automated Modeling of Physical Systems in the Presence of Incomplete Knowledge. Ph. D. thesis. University of Texas at Austin.

Farquhar, A. (1994). A Qualitative Physics Compiler. Proceedings, Twelfth National Conference on Artificial Intelligence, Seattle, Washington, The AAAI Press/The MIT Press.

Forbus, K. D. (1984). "Qualitative Process Theory." Artificial Intelligence **24**(1-3):

Genesereth, M. R. and Fikes, R. E. (1992). Knowledge Interchange Format, Version 3.0 Reference Manual. Technical report Logic-92-1, Stanford University Logic Group.

Low, C. M. and Iwasaki, Y. (1993). "Device modelling environment: an interactive environment for modelling device behaviour." Intelligent Systems Engineering **1**(2): 115-145.

Steele, G. L. (1990). Common Lisp: The Language. Digital Press.