Proceedings of the 2006 IEEE
Conference on Computer Aided Control Systems Design
Munich, Germany, October 4-6, 2006

Unit & Dynamic Typing in Hybrid Systems Modeling with CHARON

Madhukar Anand, Insup Lee, George Pappas and Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
{anandm,lee,sokolsky,pappasg}@cis.upenn.edu

***Abstract*— In scientific applications, dimensional analysis forms a basis for catching errors as it introduces a type-discipline into the equations and formulae. Dimensions in physical quantities are measured via their standard *units*. However, many programming and modeling tools provide limited support for incorporating these units into the variables. Thus, it is quite difficult for a programmer to ensure dimensional consistency in the code. Different existing standards for units further complicates this problem and an incautious use could cause inconsistencies, often with catastrophic results.**

In this paper, we propose an extension of the basic type system in CHARON, a language for modeling of hybrid systems, to include Unit and Dynamic data types. Through a combination of indirect user-guided annotations and type-inference, we address the problem of ensuring both dimensional consistency, and consistency with respect to different unit-systems. Further, we also introduce dynamic data typing, that allows programmers to specify entities that bind at runtime. Such abstractions are particularly useful to program controllers for dynamic environments. We illustrate these benefits with an example on mobile robots.

I. INTRODUCTION

When dealing with physical quantities, dimensional analysis is an effective tool to check for compatibility and plausibility. Arithmetic operations on different quantities are only meaningful if they are of the same dimensions. Any assignment or comparison also involves quantities with similar dimensions. It is therefore imperative that, programming languages and modeling tools allow specification of variables in appropriate units and incorporate dimensional analysis into the type-checking. Surprisingly however, the language support for unit-types is quite limited, often requiring explicit annotations which can be quite tedious. As a consequence, programs are frequently developed with outputs having unexpected units [6].

Including unit-types in the language alone does not solve the problem. Modular programs implemented with mixed units (c.f., SI [18], FPS[16], etc.) can harbor serious bugs and can have disastrous consequences. For example, an on-ground system used in the navigation of the NASA Mars Climate Orbiter spacecraft failed to convert between pound-seconds and newton-seconds in calculating the impulse produced by thruster firings, due to a programmer error when updating a program used for a previous spacecraft to include the specification of a new model of thruster [9]. Another well known example is from that of the ARIANE 5 flight 501 disaster. An internal software exception responsible for the failure of the launcher was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This triggered an exception in software and subsequently the flight failure [17]. As these examples illustrate, it is essential that we focus on providing support for units and their compatibility.

This research is supported in part by NSF CCR-0209024, NSF CNS-0410662, NSF CNS-0509327, and ARO DAAD19-01-1-0473

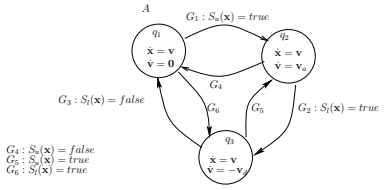

Fig. 1. Obstacle Avoidance Controller Model for a Mobile Robot

While ensuring consistency of units and types is indispensable in general, for hybrid systems, there is also a need for *dynamic* types. Many embedded systems, like that of sensor networks and mobile agents, are best modeled as a hybrid system(s). Such applications are characterized by (1) dynamic membership of a set of attributes, and (2) attributes varying in their type and units. These requirements warrant the introduction of *dynamic* type checking. The following example highlights these issues.

Example 1: Consider the case of a mobile robot in a two dimensional plane. Suppose that the objective of the application is to move the robot while avoiding and maintaining a safe distance from obstacles. A simple hybrid controller for the robot is described as in Figure 1. The 3 states of the controller correspond to whether robot is moving with constant velocity (q_1), accelerating (q_2), or decelerating (q_3). The transition conditions are described by the conditions S_u and S_l which are defined as:

$$S_u(\mathbf{x}) = \begin{cases} \texttt{true} & \text{if } \forall O_i \in \mathcal{O}, d(\mathbf{x}_i, \mathbf{x}) > d_{max} \ ; \\ \texttt{false} & \text{otherwise;} \end{cases}$$

$$S_l(\mathbf{x}) = \begin{cases} \texttt{true} & \text{if } \exists O_i \in \mathcal{O}, d(\mathbf{x}_i, \mathbf{x}) < d_{min} \ ; \\ \texttt{false} & \text{otherwise;} \end{cases}$$

where $\mathcal{O}$ denotes the *current* set of obstacles detected, $\mathbf{x}_i$ denotes the location of obstacle o_i, and d_{max}, d_{min} are safety parameters.

If the set of obstacles is updated based on readings sent in by different cameras embedded in the environment,ensuring that the data has the same type and units has to be performed at runtime. Furthermore, an abstract data array whose mem-

bership can be updated at runtime is useful in modeling many applications. For instance, with such a dynamic array type, the set of obstacles ($\mathcal{O}$) is easily modeled. □

The rest of the paper is structured as follows. In Section II, we present an overview of CHARON. In Sections III and IV we present the unit and dynamic type extensions to CHARON and discuss resolution of types and units. Finally, we present related and prior work and conclude in Section VII.

II. OVERVIEW OF CHARON

CHARON is a language for modular specification of interacting hybrid systems based on the notions of agent and mode. For hierarchical description of the system architecture, CHARON provides the operations of instantiation, hiding, and parallel composition on agents, which can be used to build a complex agent from other agents. The discrete and continuous behaviors of an agent are described using modes. For hierarchical description of the behavior of an agent, CHARON supports the operations of instantiation and nesting of modes. Furthermore, features such as weak preemption, history retention, and externally defined Java functions, facilitate the description of complex discrete behavior. Continuous behavior can be specified using differential as well as algebraic constraints, and invariants restricting the flow spaces, all of which can be declared at various levels of the hierarchy. The modular structure of the language is not merely syntactic, but also reflected in the semantics so that it can be exploited during analysis.

The key features of CHARON are its architectural and behavioral hierarchy, and constructs for discrete and continuous update of variables [2].

Example 2: The following code snippet shows how CHARON model of the obstacle avoidance controller from Example 1. The controller has three locations labeled ConstantVel, Accel, and Decel. The two continuous variables are the velocity (v) and the position (x). The mode TopMode captures the entire model. This mode is composed of the three modes. The code for mode ConstantVel is also given below. Here, the rate of change of position and velocity are captured by specifying the differential equation associated with it ($\dot{x} = v$, and $\dot{v} = 0$). The transitions between states are specified in the TopMode as shown.

```
mode TopMode (real x1, real v1){

  write analog x,v;
  mode q1= ConstantVel();
  mode q2= Accel();
  mode q3= Decel();

  trans from default to q1 when true do {x=x1;v=v1;}
  trans from q1 to q2 when (Su(x)= true) do {}
  trans from q2 to q1 when  (Su(x)= false) do {}
  ...
}

mode ConstantVel()
{
 write analog real x,v;

 diff {d(x)== v; d(v)==0}
 inv{Su(x)=true}
}
```

□

III. UNIT TYPES

In the proposed extensions to the basic typing in CHARON, we support two kinds of declarations for unit-systems: (1) using the standard set of units provided in the library, and (2) custom declaration of unit-systems using the `unit` directive.

The programmer can specify to use standard unit-systems that are built-in by using the `UnitSystem` declaration. For example, `define UnitSystem=SI` would indicate that all the physical quantities to be used will have SI units. We plan on supporting other popular unit systems such as the CGS, FPS, and the US metric standards [16]. By making the declaration of unit system global, we disallow mixing of different unit-systems, which is essential in ensuring consistency. However, this restriction only applies to a *single* program or module. Different modules can, in principle, have different unit-systems. This is clearly advantageous if the program is being developed as different modules, possibly by different people. If this be the case, there needs to be a mechanism to reconcile the different systems in use. We will come back to this problem later in Section IV.

In order for the type checker to automatically assign units under this scheme, we allow variable declaration in terms of basic physical quantities. These physical types (*pType*) will be derived from the base type `real`. This admits inference of both the *base* type and the unit for the variable. The following example clarifies the usage.

Example 3: Consider the example of the robots from Example 1. Let us say that we want to declare the displacement of the robot **x** in the SI units. This can be declared as,

```
define UnitSystem=SI;
length x[2];
```

By declaring the variable `x` to be of the type `length`, the programmer specifies both the base type (`Real`) and also its unit (meters). □

We define the 7 basic physical types (*pTypes*) in CHARON based on the fundamental physical quantities: $\mathcal{P}^0 = \{$*Length, Mass, Time, Temperature, Current, Amount of a substance, Luminous intensity*$\}$. Any other physical quantity can then be described using these basic *pTypes*. Each of these *pTypes* will have an associated unique unit which is decided by the programmer. Since all reasoning is based on dimensional analysis, $pTypes$ will form the basis of type-checking here rather than the units. Therefore, we will use the terms $pTypes$ and unit-types interchangeably in the discussion.

In CHARON, new $pTypes$ can be defined using the declaration `define pType` l_1 `= pType_expr`, where `pType_expr` is an arithmetic expression on basic physical types.

Example 4: In the robot example (Ex. 1), the programmer can define velocity as a derived physical type using Velocity=Length/Time. Therefore,

```
define UnitSystem=SI;
define pType velocity = length/time;
```

will allocate the variable `v` to be inferred as an floating point array of size two having the units of meters/sec. □

The alternative to using predefined unit-system is to custom declare it. In this case, the programmer can specify the units she chooses to use. The alternate system of units can be defined by using the `unit` keyword. However, the programmer can only specify the units of a basic type in $\mathcal{P}^0$. This is to disallow mixing of different units of measurement within the same program. In a few cases, a combination of units can be given a specific name. For instance, one Newton of force is equivalent to one $kg \cdot m/s^2$. To permit use of such units, the keyword `equiv` can be used after `unit` to indicate that the units are equivalent. The following examples illustrate the use of types and keywords introduced here.

Example 5: If the displacement in Example 1 is to be measured in kilometers and time in hours instead of meters and seconds respectively, it can be done using,

```
define unit length kilometers;
define unit time hours ;
define pType velocity = length/time;

length x[2];
velocity v[2];
```

This will cause length to be interpreted in `Kilometers`. Consequently, `velocity` will automatically be interpreted in Kilometers/hour. □

Example 6: Suppose a programmer has declared the units of mass, length, and time to be *kg*,*m* and *s* respectively. Now if she wishes to use and declare units for force, she could do so with the following set of commands.

```
define pType force = mass*length/(time)^2
define unit equiv force newton
```

By using `equiv`, the programmer lets the type checker know that `newton` is equivalent to the default set of units, which in this case is $kg \cdot m/s^2$. □

Although the programmer can specify her own units using `unit` keyword, CHARON has no way to know the semantics of it. This means that a declaration `unit length Km` would work in the same way as `unit length Kilometers` in Example 5. We introduce the directive $unit_1$ `unitto` $unit_2$ `(var) = unit_expr;` to explicitly tie the custom-defined units with the standard set of units.

Example 7: In the Example 5, the conversion to SI units can be specified as,

```
define unit time hours;
hours unitto seconds(y) = 3600*y;
seconds unitto hours(y) = y/3600;
```

Note that we have conversions from seconds to hour and vice versa. This is important to ensure that interoperability between modules written in different units. □

The CHARON type checker will verify whether *both* directions of conversion are specified.

Not all variables in a CHARON program will be physical quantities. For instance, booleans and some constants have no physical units. To deal with these quantities, we introduce a `nodim` type that can be assigned to variables that are dimensionless, and consequently without a unit. All types other than `real` such as booleans will also be assigned this unit type. A programmer can also explicitly declare floating point quantities to be `nodim` or can annotate them after the type-inference procedure is completed. This idea is explained further in Section III-B.

It must be emphasized that the unit-types are built on the regular type system and this prevents the system from having to qualify ill-formed unit-types. For instance,if 3 is declared as a constant with no dimensions, $3 * Hello$ would be assigned a unit-type of length (Sec. III, Table I). However, this would fail the regular type check and thus would not be termed as a permissible expression.

A. Unit Type Resolution

In the previous section, we have introduced the unit types in CHARON and the syntax to specify them in programs. Here, we describe the process of type checking the unit types and the procedure for type inference that we use to deduce the unit types of dimensionless quantities and constants. The Table I gives the syntax and the typing rules for a subset of the CHARON language. The unit-type checker applies the rules to check whether the expressions have consistent unit-typing after the regular type checker is done checking the program.

In the table $\mathcal{P}$ denotes the set of all physical types, ξ is a mapping that maps the physical types to their units and Γ is a typing context, i.e., the context that keeps track of the physical types of variables and expressions in the program. The table is presented as a sequence of inference rules that the type checker uses. Therefore, a rule of the form $\frac{A}{B}$ would mean that, upon seeing a declaration A, our inference of the typing would be as specified in B.

The rules $T_1 - T_5$ give the typing rules for a subset of variable declarations. T_1 deals with the declaration of a new physical type. In this case, the set of physical types $\mathcal{P}$ is augmented with the new type in the declaration (θ) and the units of the new type are set to the units of the evaluation of the typed expressions. For example, `define pType velocity = length/time` would cause `velocity` to be assigned the quotient of default units of length over time. TE represents a regular expression on typed restricted to a product or quotient of other physical types (or consequently, types raised to integer powers). Addition and subtraction of two physical types is only meaningful when both the summands are of the same physical type. Therefore, there will never be the case that a sum/difference of physical types yields a new type and we can safely ignore these operations on the typed expressions.

The `unit` declaration causes the mapping ξ to be updated (T_2) and other datatypes (bool, int, etc.) are assigned the unit-type of `nodim` (T_3). The rules T_4 and T_5 are concerned with typing of arrays. The notation $v.[] : \theta$ denotes θ to be the type of the variable array.

Typing rules $T_6 - T_{10}$ are rules for inference of types. The essence of the rules is that, sum, difference, comparison and assignment of expressions should have the same unit-type. In these rules, E represents a regular expression on variables involving common arithmetic operations. In case of differential equations, the variable and the expression

Typing Rules:

T_1 : *New physical types*

$$\frac{\texttt{pType}\ \theta\ = TE}{\mathcal{P} = \mathcal{P} \cup \{\theta\}, \xi(\theta) := \xi(\|TE\|)}$$

T_2 : *New units*

$$\frac{\texttt{unit}\ \theta\ u}{\xi(\theta) := u}$$

T_3 : *Other datatypes*

$$\frac{\theta\ v, \theta \notin \mathcal{P}}{\Gamma = \Gamma \cup \{v : \texttt{nodim}\}}$$

T_4 : *Arrays*

$$\frac{\theta\ \texttt{array}\ v[size], \theta \in \mathcal{P}}{\Gamma = \Gamma \cup \{v.[] : \theta\}}$$

T_5 : *Array elements*

$$\frac{\Gamma \vdash v.[] : \theta, \alpha : nodim}{\Gamma \vdash v[\alpha] : \theta}$$

T_6 : *Sum/Difference of expressions*

$$\frac{\Gamma \vdash E_1 : \theta, E_2 : \theta}{\Gamma \vdash E_1\ \oplus\ E_2 : \theta} \qquad \oplus \in \{+, -\}$$

T_7 : *Product/Quotient of expressions*

$$\frac{\Gamma \vdash E_1 : \theta_1, E_2 : \theta_2}{\Gamma \vdash E_1\ \otimes\ E_2 := \theta_1 \otimes \theta_2}$$
$$\otimes \in \{*, /\},\ \theta_1 \otimes \texttt{nodim} = \theta_1, \texttt{nodim} * \theta_2 = \theta_2$$
$$\texttt{nodim}/\theta_2 = (\theta_2)^{-1}, \texttt{nodim}^{-1} \stackrel{def}{=} \texttt{nodim}$$

T_8 : *Exponentiation of expressions*

$$\frac{\Gamma \vdash E_1 : \theta, c : nodim}{\Gamma \vdash E_1\ \hat{}\ c : \theta^{\|c\|}, \texttt{nodim}^{\|c\|} \stackrel{def}{=} \texttt{nodim}}$$

T_9 : *Assignment and comparison*

$$\frac{\Gamma \vdash v : \theta, E : \theta}{\Gamma \vdash v\ \prec\ E : \texttt{nodim}} \qquad \prec \in \{:=, =, \neq, <, >, \geq, \leq\}$$

T_{10} : *Differential equations*

$$\frac{\Gamma \vdash v : \theta_1, E : \theta_2, \xi(\theta_2) \cdot \xi(time) = \xi(\theta_1)}{\Gamma \vdash \texttt{d}(v)\ =\ E : \texttt{nodim}}$$

TABLE I
UNIT TYPING RULES FOR CHARON

differ in the unit of time. For product and quotient of expressions, there are a few cases depending on whether one of the expressions is of type `nodim`. The rules check for consistency of the physical types, and since the physical types have a *unique* unit throughout the program, their units will also be consistent.

Before we close the discussion on typing rules, we mention that we have not considered type casts with the exception of casting constants to a particular physical type. This restriction is placed so that programmers cannot arbitrarily cast one physical type to another. It can be argued that there are very few situations, if any, where such type casts would be meaningful. However, we leave the possibility open, of allowing limited class of such explicit casts in future.

B. Automatic Inference of Unit-types

The typing rules introduced in the previous section will fail to check equations and assignments where only some of the quantities have a declared physical types. Examples of this could include constants which are not explicitly declared. Indeed, depending on context, constants such as 0 could have multiple unit-types. Instead of imposing that all of them be explicitly annotated, we propose to automatically infer the types of unknown quantities and prompt the programmer to annotate them only on a reduced set of constraints. This process is similar in principle to the annotation-less type inference approach in [10] but differs in being semi-explicit (some units are known) and the constraints generated are in terms of physical dimensions rather than units. The inference process is done in two phases. The first phase involves constraint generation and the second phase involves solving these constraints.

Constraint generation: In this phase, the code is first analyzed and the expressions whose type cannot be resolved using the rules $T_7 - T_{10}$ are assigned an variable $pType$ of t. Typically, the equations that are not resolved by the checker include ones that have implicit constants multiplied in them. For instance, $x = \sqrt{v}$ is one such equation. Here, the constant 1 on the right hand side has dimensions of $(Length)^{\frac{1}{2}}(Time)^{\frac{1}{2}}$. To cope with such expressions, we include a fresh unit-type t with every equation that either has a mismatched unit or one which lacks constants. This step is not essential for equations which has constants of unspecified type. such as the equation $x = 3\sqrt{v}$ where the constant 3 can be assigned the appropriate dimensions. Finally, we equate the dimensions on both sides of the equations to generate the constraints. The following set of examples illustrates the procedure.

Example 8: Consider the equation $x = \sqrt{v}$. The left hand side of this equation has dimensions of $(Length)$ while the right hand side has dimensions $(Length)^{\frac{1}{2}}(Time)^{-\frac{1}{2}}$. To ensure equality, we introduce a constant 1 and denote its type to be t_1. With this constant, the type equality requirement now becomes, $L = t_1 * L^{\frac{1}{2}} * T^{-\frac{1}{2}}$ where L denotes the length and T the time. Equating the dimensions of L and T on both sides, we can get the following equations : $1 = t_1^L + \frac{1}{2}$, and $0 = t_1^T - \frac{1}{2}$. Solving these equations yields the type (and hence the units) of the constant to be $(L)^{\frac{1}{2}}(T)^{\frac{1}{2}}$. □

The unit-type checker makes it run through the program and automatically creates constraints for expressions that do not unit-type check.

Example 9: Consider the following code snippet implementing the robot example (Ex. 1) in location q_1: Let us assume that x is declared to be one-dimensional and of physical type *length* but v is only declared as `real`. The type constraints generated are indicated as comments.

```
real analog velocity;

diff(d(x)==v);          // L = t_0 * t_1 * T
diff(d(v)==0);          // t_1 = t_2 * T
```

Here L and T refer to the length and time dimensions used in dimensional analysis. t_0 is the type assigned to the implicit constant 1, t_1 is the type associated with variable v and t_2 is the type associated with the constant 0. We get the following constraints from these: $t_0^L + t_1^L = 1$ (Equating *Length* in the first equation), $t_0^T + t_1^T + 1 = 0$ (Equating *time*), and similarly, $t_1^L = t_2^L$ and $t_2^T + 1 = 0$ from the next equation. □

The generated constraints can be solved using standard

equation solvers. In case the system is such that there are more variables than equations, the programmer is prompted to provide the minimum number (Number of equations - number of variables) of missing types of variables.

IV. Dynamic Types

$$T_1^{'}: \textit{Dynamic arrays} \quad \frac{\theta \ \texttt{dynArray}\ var, \theta \in \mathcal{P}}{\Gamma = \Gamma \cup \{var.get() : \theta\}}$$

$$T_2^{'}: \textit{dTypes} \quad \frac{\texttt{define dType var(T x,...)= bf;}, \Gamma \vdash \texttt{bf:bool}}{\mathcal{D} = \mathcal{D} \cup \{var <: T\}}$$

$$T_3^{'}: \textit{Runtime Unit Typing} \quad \frac{\Gamma^{'} \vdash \texttt{varhandler} : \theta, \theta \in \mathcal{P}, \xi^{'}(\theta) = \xi(\theta)}{\Gamma \vdash \texttt{extern handler (var, varHandler)} : true}$$

TABLE II

Partial set of typing rules for dynamic types in Charon

In addition to unit types, an orthogonal but useful abstraction for modeling many hybrid controllers is that of a *dynamic* type. There are two flavors to dynamic typing. In the first case, we consider data structures that have a constant unit-type but are updated *externally* and therefore it is essential to ensure that the writes to the variable by the external agent be of the same type and units.

In the second kind of dynamic types, we consider variables that can acquire different abstract *types* at runtime. Such dynamic types are defined as a subtype relation on static unit-types. This implies that, while variables have the same $pType$ (and hence the units) throughout the program, they can belong to multiple dynamic types. For instance, in the robot example (Ex. 1), we can define *Obstacles* to be of *pType* length, and then either to be of type S_l or S_u depending on the relative distance from the robot. The main advantage of such abstractions is that they separate the *update* definition and *use* of the variables.

The syntax for specifying these dynamic types is,

```
pType dynArray var;
extern handler (var, varHandler);

define dType var(T x,T1 V1,...)= bf;
```

Since Charon is built on Java, we use the `vector` object (`java.util.Vector`) to construct the dynamic array. We cannot use the `vector` class directly as it allows allows two different types of objects to be added. We remedy this by having `dynArray` restrict member updates to be only of type $pType$. Apart from this restriction, all other methods inherited from the `Vector` are provided for use. Examples of these methods include, `add, delete, isEmpty, get`, etc. The entire list of methods for the `Vector` class can be found in the Java documentation page [12].

The keyword `extern` can be used to declare an external function that updates the members of the array. This declaration has the effect that it lets the runtime know that the unit-type and unit-system check has to be performed before the function can update the members.

In the syntax for specifying dynamic type *dType*, `bf` is a boolean function on variables `V1,...,Vn` and `x`. If the function `bf` evaluates to true, then `x` will be assigned a type `var`. The type `var` here, is treated as a subtype of type `pType`.

Example 10: Consider the problem of mobile agents moving in the plane with the same speed but variable heading direction. Each agents heading is updated as the average of its heading and a set of its nearest neighbors. As the agents move, the graph induced by the nearest neighbor relationship changes, resulting in switching. In this case, you want to ensure that only the nearest neighbors' data is read at any agent. We can enforce this check by introducing a dynamic type to handle the situation. Here is a code snippet with the type declaration.

```
define dType near(Cd xn, Cd xs) = {
        if(d(xn,xs)<dmin)return true;
        return false;
}
near dynArray N;
```

It is assumed that `xn` are the 2-dimensional coordinates (`Cd`) of a neighbor and `xs` are the co-ordinates of the agent and `d` returns the distance between different co-ordinates. If the distance between them is less than `dmin`, then the neighbor qualifies as being `near`. □

A. Inference of Dynamic Types

The Table II gives a partial list of typing rules for the abstract types introduced here. The rule $T_1^{'}$ gives the typing for arrays. The effect of declaring a dynamic array is the same as that of static arrays except that now the `var.get` method will be expected to return with a value of type as declared. The rule $T_2^{'}$ describes the typing for declaration of a new *dType*. The effect of the declaration is that, the variable `var` is inferred to be a subtype of the type `T` and it is added to $\mathcal{D}$, the set of all $dTypes$. The notation $<:$ indicates the sub-typing.

Finally, the rule $T_3^{'}$ describes type-checking of external binding. This rule is different from the rest, in that, it is interpreted at runtime. Therefore, a runtime support needs to be provided to the Charon compiler and type-checker. While this may not be possible in general, a wrapper can be built around each Charon program that implements this check for every input and output. A discussion on runtime support for dynamic types in Java can be found in [15]. We plan to follow up on that, and develop a runtime support for Charon in future. Nevertheless, with such a system in place, the external context $\Gamma^{'}$ and mapping function $\xi^{'}$ can be called to check whether the quantities returned by the external handler have the same units. In case of discrepancies, the runtime can then use `unitto`, if it is defined, to supply the value of the variables in the right units.

V. Related and Prior Work

There are several modeling tools for hybrid systems such as Charon [2], Ptolemy [7], Shift [8], and the Matlab/Simulink Hybrid Toolbox [11]. A listing of many tools and their description is available at [20]. Ptolemy has a

provision to incorporate units. Currently, their system has the SI units hard-wired and it does not yet support automatic unit-type inference and dynamic types like `dType`.

Adding unit information to programming languages has been the topic of frequent research. Languages such as C++ have powerful extensions that the unit type information can be added within the language with concepts such as overloading and templates [6]. In other languages, such as Java, units have been treated as a class [1]. There have also been a few scientific tools that consider units such as a unit-checking tool for Microsoft Excel Spreadsheets [3] and Unit extension for FORTRAN [19]. In many modeling tools such as Modelica [4], dimensions and units are a part of the language specification. However, there exists no strategy to analyze and verify dimensional integrity in a arbitrary language construct of Modelica and there is an ongoing project to incorporate them [5].

A common approach to adding unit types is to let the user annotate quantities with their appropriate units. This scheme however requires all quantities such as constants have explicit units and therefore can be quite tedious. To reduce the burden of annotations, researches have suggested embedding unit types in a type system like that of ML that supports type inference [21], [13]. Kennedy [13] implements a algebraic technique that allows only integral exponents and also gives theoretical results on the expressiveness of such a system. A annotation-less unit type inference for C has been suggested in [10]. Although such a technique does not require annotations, it suffers from imprecision as the tool is context-insensitive and the constraint solving can become a bottleneck if the programs to be checked are large.

In contrast to the above, our approach utilizes an indirect annotation of the units. This is accomplished by supporting physical quantity types such as `length` and `temperature` and specifying the unit-system to be used. Type inference is used to check consistency in differential operators and of constants. The dynamic types are managed with the help of a runtime environment that evaluates and decides the types of objects at runtime.

VI. Implementation Outline

The implementation of unit and dynamic type checking is orthogonal to the regular type checking in CHARON . After the type checker returns successfully, the code has to be parsed for units and generating dimensional constraints. Java tools such as JavaCC can be used to automate the constraint generation. The next phase involves solving the unit typing constraints and this can be done by invoking any equation solver. For dynamic types, we need a runtime support for CHARON. Specifically, we need to instrument the code to monitor the changes and then tie the changes to the types. We plan on using several concepts from the Java runtime monitoring and control, Java-MAC [14] to implement the dynamic typing.

VII. Conclusions

In this paper, we have described a framework to incorporate unit types to verify and ensure consistency in scientific modeling tools such as hybrid systems. We have presented typing rules for unit types based on indirect user-guided annotations and automatic inference. We have also proposed a scheme to include runtime checking of units that would help in automatically convert between different unit-systems. We are currently working on implementing these features in CHARON.

Acknowledgments. We would like to thank Gunjan Gupta, and anonymous reviewers for their suggestions in improving the paper.

References

[1] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Jr. Guy L. Steele. Object-oriented units of measurement. In *19th annual ACM Conference on Object-oriented programming, systems, languages, and applications*, pages 384–403, 2004.

[2] Rajeev Alur, Radu Grosu, Yerang Hur, Vijay Kumar, and Insup Lee. Modular specification of hybrid systems in CHARON. In *HSCC*, pages 6–19, 2000.

[3] Tudor Antoniu, Paul A. Steckler, Shriram Krishnamurthi, Erich Neuwirth, and Matthias Felleisen. Validating the unit correctness of spreadsheet programs. In *Proceedings of the 26th International Conference on Software Engineering*, pages 439–448, Washington, DC, USA, 2004. IEEE Computer Society.

[4] Modelica Association. Modelica specification, version 2.2. `http://www.modelica.org`, February 2005.

[5] David Broman. Static dimensional analysis in modelica. `http://www.ida.liu.se/~davbr/dim-analysis.pdf`.

[6] Walter E. Brown. Applied template metaprogramming in SI units: the library of unit-based computation. In *Second Workshop on C++ Template Programming*, 2001.

[7] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. pages 527–543, 2002.

[8] A. Deshpande, A. Gll, and L. Semenzato. The SHIFT programming language for dynamic networks of hybrid automata.

[9] Edward A. Euler, Steven D. Jolly, and H.H. 'Lad' Curtis. The failures of mars climate orbiter and mars polar lander: A perspective from the people involved. In *24th Annual AAS Guidance and Control Conference*, 2001.

[10] Philip Guo and Stephen McCamant. Annotation-less type inference for C, pag.csail.mit.edu/6.883/projects/unit-type-inference.pdf, 2005.

[11] Hybrid Toolbox - Hybrid Systems, Control, Optimization. http://www.dii.unisi.it/hybrid/toolbox.

[12] Java 2 Platform SE Class Vector. http://java.sun.com/j2se/1.3/docs/api/java/util/vector.html.

[13] Andrew J. Kennedy. Relational parametricity and units of measure. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 442–455, New York, NY, USA, 1997. ACM Press.

[14] Moonjoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[15] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 337–361, London, UK, 2000. Springer-Verlag.

[16] NIST. Preferred metric units for the general use by the federal government, 1993.

[17] Bashar Nuseibeh. Ariane 5: Who dunnit? In *IEEE Software Vol. 14, Issue 3.*, pages 15–16, 1997.

[18] International Bureau of Weights and Measures (BIPM). The international system of units, 1998.

[19] Grant W. Petty. Automated computation and consistency checking of physical dimensions and units in scientific programs. *Softw. Pract. Exper.*, 31(11):1067–1076, 2001.

[20] Hybrid System Tools. http://wiki.grasp.upenn.edu/ graspdoc/hst/.

[21] Mitchell Wand and Patrick O'Keefe. Automatic dimensional inference. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 479–483, 1991.