Compatibility and reuse in component-based systems via type and unit inference

Christian Kühnel[1] Andreas Bauer[2,1] Michael Tautschnig[1]

[1)]Institut für Informatik, Technische Universität München
[2)]National ICT Australia (NICTA), Canberra

Abstract

In many branches of industry, the component-based approach to systems design is predominant, e. g., as in embedded control systems which are often modelled using MATLAB/Simulink. In order to facilitate reuse, and to raise the level of abstraction for future designs and frequently used functions, the employed tool sets offer built-in mechanisms to create sophisticated component libraries. For large, real-world designs, however, it is not always clear, whether or not a certain context violates even the most basic design assumptions of employed library components, thus often leading to expensive runtime errors. This paper introduces a practical method for checking compatibility of large designs, statically. This method not only ensures that large component-based designs provide a context such that all (library) components have well defined types, but it also ensures that transmitted physical units, such as $m^2, km/h, mph$, etc. are preserved during computation. As such the possibility for runtime errors is reduced, and a metric for sound component reuse given.

1 Introduction

Component-based approaches to software and systems design are predominant in large parts of industry, such as automotive and embedded control systems in general. Tool chains used in these domains are often based on MATLAB/Simulink, a graphical modelling and simulation environment with the ability to generate code (via program extensions). MATLAB/Simulink and other tools further support the creation and use of dedicated component libraries such that frequently used functionality can be reused in different contexts, and without having to concentrate on implementation details. As such, the reuse of components raises the level of abstraction for many designs, and development time is often reduced.

However, this convenience comes at a price; that is, the developer has to make sure that reused components are compatible in the context they are used in. For large, real-world designs as they are common, e. g., for automotive control systems like adaptive cruise control, engine management, or electronic brake systems, reuse is often constrained by the ability to establish firm criteria for ensuring compatibility of reused components with respect to the rest of a complex design, i. e., a *context*.

Contribution. In this paper, we focus on establishing a criterion for *syntactic compatibility*, i. e., a criterion that can be checked at compile-time and without executing the component-based design under scrutiny. In the context of some of our industrial research collaborations, compatibility checking has been reduced to a problem of *type inference* for which a comprehensive theory already exists (cf. [13]), and which can be greatly automated and tool-supported. We say that a given design is compatible w. r. t. its employed library components iff (1.) a well-typed design can be inferred from it, and (2.) the inferred types do not violate a set of predetermined *measurement units*. Measurement units (see Sec. 4) play an important role in domains where systems control physical processes, e. g., for ensuring that signals are interpreted correctly by another system or component. Therefore, a *decidable polymorphic* type system for a custom component language is introduced that can be used to infer for a concrete design its types and according measurement units, and determine compatibility. However, this type system and inference algorithm is not restricted to our language, but can be applied to similar component-based formalisms, e. g., MATLAB/Simulink at ease.

In the context of our current projects, our approach to compatibility testing has proven to be a reliable metric not only in terms of establishing type compatibility of large designs and component libraries, but also as a means of avoiding type errors in the running program and the code generator of our component language. An implementation of our method would exhibit only linear runtime in the number of design artifacts due to the use of standard *unification* algorithms (cf. [9]) for inference. Moreover, not only type correctness can be established formally, but also compatibility between the measurement units of a design.

Related work. Most tools which support a component or model-based design approach, and which are currently used in the domain of embedded control systems, do not offer advanced concepts for type checking, let alone type inference. The authors of this text believe that the reason lies in that these systems target rather low-level design artifacts, such as the design of continuous control algorithms, and abstraction from this level of detail has not been the primary focus of the developers for a long time.

The concept of measurement units in combination with a strong static type system, however, caters for comprehensible designs, and avoids errors in the actually running systems. Moreover, they facilitate the reuse of components, and give to the user a powerful tool for combining different library components in different contexts without violating assumptions over measurement units or data types, and not being aware of it.

Currently, MATLAB/Simulink does not offer strong static type checking, and subsumes mostly basic data types such as `boolean` and `double` which, if not specified at "compile-time", may create runtime errors due to maltyped designs; that is, if nothing is explicitly specified for a component, then `double` is used as a "default type" for input and output signals (cf. [5, 1]). The situation is a similar one for other domain-specific solutions such as ASCET-SD.

However, various authors have concentrated on establishing *behavioural correctness* of low-level designs. A popular formalism are *Interface automata* [6]. These can be used for the specification of behaviour at component interfaces, and to establish compatibility between compound interfaces by means of automata analysis. One of the research tools already making use of this is, e. g., Ptolemy II [11]. The static part of Ptolemy's type system, however, is rather different from the approach presented in this paper, in that the behaviour of Ptolemy II components (called "actors") is tightly interwoven with Java code. Therefore, type correctness is partly handled by the Java compiler, and Java is known to leave space for subtle type errors at runtime.

Our custom language, in the next section introduced as SCL, handles type inference at compile time and avoids such problems altogether. To the best of the knowledge of the authors, no other component-based design language for the development of embedded control systems currently offers the combination of polymorphic type and measurement unit inference as well as a decidable, static type system.

2 SCL—A simple component language

In the following, we introduce a simple component language, SCL, that provides the essential mechanisms for systems modelling and analysis. Conceptually, SCL is close to other graphical modelling formalisms, such as UML-based ones, or MATLAB/Simulink, but its semantics is based upon a uniform discrete time-base and the hypothesis of perfect-synchrony [2]. This basically asserts that computations of components occur instantly, i. e., take no time, and that communication is infinitely fast. This allows us to abstract from implementation details when modelling, and caters for a sound and well-defined semantics. Various tools for systems design and synthesis exist that are based on this notion, e. g., SCADE [5], Esterel [3], and AutoFocus [4].

In the following the syntax and semantics of SCL are introduced. The syntax is required for the definition of the typing rules (see Sec. 3.1), the semantics for the safety proof (see Sec. 3.2).

2.1 Concepts and syntax

Basically, SCL provides the following core concepts for modelling: (functional) blocks, ports and channels for communication, (composite) components, all of which are explained in the following.

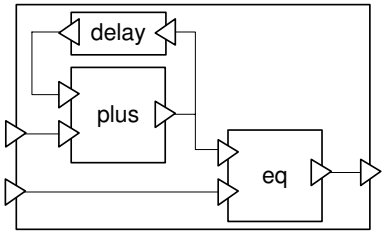

Figure 1. An example SCL model

A block is a "primitive" component whose functionality is defined by the semantics of SCL. It cannot be decomposed further. For instance, the predefined arithmetic block $plus$ simply adds two variables, which are delivered to it via two input ports (see Fig. 1). Further, a dedicated $delay$ block (i. e., z^{-1}) is used to store values for one computational cycle, and essential for avoiding causal loops in the model. Hence its value gets updated periodically, and needs to be initialised to a default value when invoked for the first time. For brevity, only some predefined blocks are discussed here.

A block has a signature S and parameters depending on the type of block, e. g., the $delay$ block $delay((p_i \rightarrowtail p_o), v)$ has an initial value v. (Composite) components, denoted $component((S), ch, C, \Phi)$, are composed of a nonempty set of subcomponents C, and a nonempty set of channels ch, and have a well-defined signature $S = p_1^{in}, \ldots, p_n^{in} \rightarrowtail p_1^{out}, \ldots, p_n^{out}$ that is made up of its outside-visible input and output ports. The type context Φ is used to introduce new type variables.

Note that a more detailed account of the syntax is available in [10].

$$\frac{\langle ord(ch \cup C), \sigma, \eta\rangle \to \langle \sigma', \eta'\rangle}{\langle component(S, ch, C, \Phi), \sigma, \eta\rangle \to \langle \sigma', \eta'\rangle} \quad \text{(S-comp)}$$

$$\frac{\langle delay_l((p_i \rightarrowtail p_o), v), \sigma, \eta\rangle \to \langle \sigma', \eta\rangle \quad \langle delay_s((p_i \rightarrowtail p_o), v), \sigma, \eta\rangle \to \langle \sigma, \eta'\rangle}{\langle delay((p_i \rightarrowtail p_o), v), \sigma, \eta\rangle \to \langle \sigma', \eta'\rangle} \quad \text{(S-delay)}$$

$$\frac{\langle e, \sigma, \eta\rangle \to \langle \sigma', \eta'\rangle \quad \langle es, \sigma', \eta'\rangle \to \langle \sigma'', \eta''\rangle}{\langle e; es, \sigma, \eta\rangle \to \langle \sigma'', \eta''\rangle} \quad \text{(S-seq)}$$

$$\frac{\sigma(p_a) = v_a \quad \sigma(p_b) = v_b \quad v_c.value = v_a.plus(v_a.value, v_b.value)}{\langle plus(p_a, p_b \rightarrowtail p_c), \sigma, \eta\rangle \to \langle \sigma[v_c/p_c], \eta\rangle} \quad \text{(S-plus)}$$

$$\frac{\sigma(p_i) = v_i}{\langle delay_s((p_i \rightarrowtail p_o), v), \sigma, \eta\rangle \to \langle \sigma, \eta[v_i/p_i]\rangle} \quad \text{(S-delay-store)}$$

$$\frac{\eta(p_i) = \bot}{\langle delay_l((p_i \rightarrowtail p_o), v), \sigma, \eta\rangle \to \langle \sigma[v/p_o], \eta\rangle} \quad \text{(S-delay-nohist)}$$

$$\frac{\eta(p_i) = v_o}{\langle delay_l((p_i \rightarrowtail p_o), v), \sigma, \eta\rangle \to \langle \sigma[v_o/p_o]; \eta\rangle} \quad \text{(S-delay-hist)}$$

$$\frac{\sigma(p_1) = v}{\langle chan(p_1 \rightarrowtail p_2), \sigma, \eta\rangle \to \langle \sigma[v/p_2], \eta\rangle} \quad \text{(S-chan)}$$

$$\frac{\sigma(p_c) = true \quad \sigma(p_t) = v}{\langle if(p_c, p_t, p_e \rightarrowtail p_r), \sigma, \eta\rangle \to \langle \sigma[v/p_r], \eta\rangle} \quad \text{(S-if-true)}$$

$$\frac{\sigma(p_c) = false \quad \sigma(p_e) = v}{\langle if(p_c, p_t, p_e \rightarrowtail p_r), \sigma, \eta\rangle \to \langle \sigma[v/p_r], \eta\rangle} \quad \text{(S-if-false)}$$

$$\frac{v = \{l_1 = v_1, \ldots, l_n = v_n\} \quad l_i \in \{l_1, \ldots, l_n\}}{v.l_i = v_i} \quad \text{(S-rcd)}$$

Figure 2. Operational semantics of SCL

2.2 Operational semantics

In order to substantiate the claims made in Sec. 1, we briefly introduce an operational semantics for SCL. This provides for a concrete type context when reasoning about type safety, and consequently compatibility of component-based designs in SCL.

The notation for the operational semantics used here is loosely based on the one presented by Plotkin [14]. The rules are displayed in Fig. 2. The current state of a system is denoted by σ, and the history by η (i. e., the previous state). This differentiation is important especially in face of the $delay$ block which stores values for exactly one computational cycle. We fix a set of ports P, and a set of channels ch, as well as a domain of values V which are transmitted via ports and channels. Let $p_1, p_2 \in P$ and $c \in ch$, then $c = chan(p_1 \rightarrowtail p_2)$ indicates that a port p_1 is connected with p_2 via channel c. Further, we use the notation $\sigma(p) \in V$ for "retrieving" a value currently associated to a port $p \in P$, and $\sigma[v/p]$ for "storing" it in the state (respectively with η). To avoid conflicts in σ and η, the identifiers of all ports and delays of a model are assumed to be unique.

Finally, we introduce an artificial function $ord : 2^E \to 2^E$, which, for a given set of entities, returns a partially ordered set of entities that defines their order of execution, where $E = C \cup ch$ and C is the set of components connected over the set of channels ch. Before a component can be evaluated, the input ports have to be assigned their values. For the system, this has to be provided by the environment, within a component this is ensured by the ord function.

In the first computational cycle (i. e., at the beginning of execution), the history is empty, $\eta = \emptyset$. In the following cycles, the history of step n is used to compute step $n + 1$.

Order of execution and causality The values of a component (S-comp) are defined according to its subcomponents and channels. The execution order of the individual subcomponents and channels is determined by function ord.

Additionally, the ord-function formally "splits" every $delay$ (S-delay) into a load part, $delay_l$, and a store part, $delay_s$. The load part is placed at the beginning of the evaluation order, and the store part at the end. This ensures that the delayed values of the last step are available to all components and that the current values of the delays are stored.

The remaining components and the channels are sorted in their evaluation order. Let $c \in C$ be some component, and let $I(c)$ denote the set of input ports of c and $O(c)$ the output ports, respectively, then a partial order $<$ is defined as follows: For two components $c, c' \in C$, and for every channel $chan(p_o \rightarrowtail p_i) \in ch$, $c < chan(p_o \rightarrowtail p_i) < c'$ iff $p_o \in O(c)$ and $p_i \in I(c')$. Since no cyclic connections without a $delay$ are allowed, $<$ delivers a partial order on model entities, returned by ord.

Let ; be the functional sequence operator. Then, an ordered sequence $e; es$ of entities is evaluated by evaluating the first element e of the sequence, delivering a new valuation sequence σ' and history sequence η'. These new sequences are then used for recursive evaluation of the remainder of the entity sequence es as depicted in rule (S-seq).

The store part of a delay takes the value $v \in V$ of the input port $p_i \in P$ and stores it in the history η for the next cycle (S-delay-store). In the first cycle, when $\eta(p) = \bot$, i. e., the port is not yet bound in η, the load part of delay block

returns its default value (S-delay-nohist). In the consecutive steps, it returns a value $v_o \in V$ stored η (S-delay-hist).

As expected, a channel simply propagates the value of one port to the other port (S-chan).

Execution of blocks. Let in what follows, $p_a, p_b, p_c \in P$ and $v \in V$. A $plus$ block is evaluated in (S-plus) by adding the values of the input ports p_a and p_b and binding the result to the output port p_c. As we are using existential types for type abstraction, (see Sec. 3.1), the $plus$-Operator of the respective type is applied. The rules for other blocks such as $minus$, $mult$, div, and, or, etc. are similar. An entry l of an existentially typed value v can be accessed by $v.l$ (S-rcd).

The evaluation of a conditional block, if, depends on the value of port p_c: if $\sigma(p_c) = true$ then the value of the port p_t is assigned to the result port p_r (S-if-true), otherwise the value of port p_e is used (S-if-false).

3 Compatibility in SCL designs

Compatibility checking in SCL is reduced to three key aspects all of which can be checked statically and automatically:

1. Type inference, i. e., inferring the appropriate types for an under-specified model, if syntactically possible.
2. Type safety, i. e., SCL ensures that a well-typed model cannot create runtime errors which are due to type failures.
3. Units, i. e., SCL models can be checked for compatibility in terms of the physical units that are computed and used in a model.

In this section, we first concentrate on 1. and 2. The introduction and elimination rules and some of the subtyping rules are omitted here for the sake of brevity, as those are similar to the standard literature (cf. [13]).

3.1 Inference of data types and signatures

When components are built for reuse, it is desirable to design them so that they can be used in a variety of contexts. For example, a sorting algorithm could be used with any type, as long as an ordering, e. g., an $\leq$-operator, is defined on that type. Another example would be a component implementing a queue: this component should be able to queue elements of a static, but arbitrary type. To implement this in the type system, *existential types*, *universal types* and *subtyping* are used.

If type S is a subtype of type T, denoted $S \sqsubseteq T$, this means that every type T in a well-typed model could be replaced with an S, and the model would still be well-typed.

(Bounded) universal types are used to introduce new type variables in the context Φ of a component (T-component). They are denoted by $\forall X \sqsubseteq T.\, X$ to express that the component can be used for any type X, which is required to be a subtype of T. This can be used to define the types of the queue example above.

Existential types are used to denote which operations have to be defined on a certain type. In combination with universal types and subtyping, the example can now be described formally: the sort component may be used with any type $\forall X \sqsubseteq T.\, X$ as long as an operation $\leq$ is defined on that type $T = \{\exists Y, \{\leq\!: Y \times Y \to \mathsf{Bool}\}\}$.

When a new type such as a record or tuple is defined by the users, they can also define operations on that type and thereby build a new existential type. This new type can now be used with any (prior) component whose type restrictions it satisfies.

SCL requires only a minimal set of primitive types, Bool and Float. Bool is required for comparison and if blocks, Float for arithmetics. Other types can be added in a similar manner. As we use only existential types throughout the model, these types are defined as:

$$
\begin{aligned}
\mathsf{Float} := \quad & \{\exists X, \quad \{value : X, \\
& \quad plus, minus, mult, div : X, X \to X, \\
& \quad sqrt : X \to X, \\
& \quad eq, le, gt : X, X \to \mathsf{Bool}\}\} \\
\mathsf{Bool} := \quad & \{\exists X, \quad \{value : X, \\
& \quad and, or : X, X \to X, \\
& \quad neg : X \to X\}\}
\end{aligned}
$$

Notably, SCL supports under-specification in models in terms of polymorphic blocks and components. For instance, a $plus$ block is polymorphic in that it works over different value domains, and its output type is determined solely by the types of input values. A model can thus be made up only of polymorphic blocks (i. e., under-specified blocks and components lacking a concrete signature), and the concrete instances are then inserted automatically by the type inference mechanism.

The (almost) complete set of typing rules of SCL is depicted in Fig. 3. A type T of a port $p \in P$ can be derived from a *type context* Γ, whenever there is exactly one occurrence of p in Γ (T-port). The set of all bound ports in Γ is defined as $dom(\Gamma)$. And whenever a typing can be derived from a context Γ', so can it from any permutation thereof (T-perm).

The ports of the $plus$ block are bound to the type T, which may be an arbitrary type for which an operator $plus$ with signature $X, X \to X$ is defined (T-plus). The rules for $minus$, $mult$, neg, and $sqrt$, etc. are similar.

For a component to be well-typed, all of its channels and sub-components have to be well-typed (T-component). A

$$\frac{p \notin dom(\Gamma)}{\Gamma, p : T \vdash p : T} \quad \text{(T-port)}$$

$$\frac{\Gamma' \vdash p : T \quad \Gamma \textbf{ permutation of } \Gamma'}{\Gamma \vdash p : T} \quad \text{(T-perm)}$$

$$\frac{\begin{array}{c}\Gamma \vdash p_a, p_b, p_c : T \\ \Gamma \vdash T \sqsubseteq \{\exists X, \{value : X, plus : X, X \to X\}\}\end{array}}{\Gamma \vdash plus(p_a, p_b \rightarrowtail p_c)} \quad \text{(T-plus)}$$

$$\frac{\begin{array}{c}\Gamma \vdash p_a, p_b : T \quad \Gamma \vdash p_c : \mathsf{Bool} \\ \Gamma \vdash T \sqsubseteq \{\exists X \{value : X, eq : X, X \to \mathsf{Bool}\}\}\end{array}}{\Gamma \vdash eq(p_a, p_b \rightarrowtail p_c) :} \quad \text{(T-eq)}$$

$$\frac{\Gamma \vdash p_a, p_b, v : T}{\Gamma \vdash delay((p_a \to p_b), v)} \quad \text{(T-delay)}$$

$$\frac{\begin{array}{c}\forall c \in C.\, \Gamma, \Phi \vdash c \\ \forall c \in ch.\, \Gamma, \Phi \vdash c\end{array}}{\Gamma \vdash component(S, ch, C, \Phi)} \quad \text{(T-component)}$$

$$\frac{\Gamma \vdash p_1 : S \quad \Gamma \vdash p_2 : S \quad \Gamma \vdash S \sqsubseteq T}{\Gamma \vdash chan(p_1 \rightarrowtail p_2)} \quad \text{(T-chan)}$$

$$\frac{\Gamma \vdash p_c : \mathsf{Bool} \quad \Gamma \vdash p_t, p_e, p_r : T}{\Gamma \vdash if(p_c, p_t, p_e \rightarrowtail p_r)} \quad \text{(T-if)}$$

$$\frac{}{\Gamma \vdash T \sqsubseteq T} \quad \text{(U-refl)}$$

$$\frac{\Gamma \vdash S \sqsubseteq T \quad \Gamma \vdash T \sqsubseteq Q}{\Gamma \vdash S \sqsubseteq Q} \quad \text{(U-trans)}$$

$$\frac{}{\Gamma \vdash T \sqsubseteq \mathsf{Any}} \quad \text{(U-any)}$$

$$\frac{\Gamma \vdash S \sqsubseteq R}{\Gamma \vdash \{\exists X, S\} \sqsubseteq \{\exists X, R\}} \quad \text{(U-ex)}$$

$$\frac{\begin{array}{c}\{k_j^{j \in 1..m}\} \supseteq \{l_i^{i \in 1..n}\} \\ k_j = l_i \to S_j \sqsubseteq T_j\end{array}}{\{k_j : S_j^{j \in 1..m}\} \sqsubseteq \{l_i : T_i^{i \in 1..n}\}} \quad \text{(U-rcd)}$$

Figure 3. Typing (T-) and subtyping (U-) rules of SCL

component is the only possibility for the user to define new type variables by adding them to the additional context Φ. The scope of these variables is the component and all its ports and sub-components.

A channel is the "link" between two components. It is well-typed, if the type of the source port $p_1 \in P$ is a subtype of the type of the target port $p_2 \in P$ as defined in rule (T-chan). Naturally a channel can be used for any type satisfying the subtype property. It is sufficient to allow subtyping only at channels, i. e., between components, as the type system would not be more expressive if we allowed subtyping within blocks.

Subtyping. The types in SCL together with the subtyping relation $\sqsubseteq$ form a lattice. The subtyping rules over this lattice are depicted in Fig. 3.

The subtyping relation is transitive (U-trans) and reflexive (U-refl) and every type is a subtype of Any (U-any). An existential type is subtype of another existential type, if their operations are in a subtype relation (U-ex). The subtyping-relation for records (U-rcd) is used for existential types, since we want

$$\{\exists X, \{value : X, plus, minus : X, X \to X\}\} \sqsubseteq \{\exists X, \{value : X, plus : X, X \to X\}\}$$

This case occurs, e. g., when one output port is connected to the input ports of a $plus$ and a $minus$ block by two channels. Then the type of that output port has to provide a $plus$ and a $minus$ operator, thus the two existential types have to be unified as defined in rules (U-ex) and (U-rcd).

These properties of the type system not only provide the possibility to check the compatibility of a component but can also be used to adjust the typing of the components to make them compatible. This is achieved by inferring a valid typing from the composed components, if such a typing is possible.

3.2 Runtime safety

Yet, establishing the compatibility of two components at compile time is not enough. The main concern is that these components remain compatible at runtime. This means that the type system must ensure that no type errors occur at runtime. We therefore have to prove *safety* of our static type system. This means that a model, which was well-typed at compile-time, remains well-typed during runtime.

In general, to show safety for a type system it is sufficient to show *progress* and *preservation* [13]. Progress means that the evaluation, based on the operational semantics, for any well-typed entity is not stuck, i. e., a port can be evaluated and for all entities, there is a rule that can be applied. Preservation means that every well-typed entity remains well-typed after an evaluation step in the operational semantics. Syntactic correctness of the model is presumed here.

Lemma 1 (Progress) *If the model and σ and η are well-typed, then every port can be evaluated and for each entity there is a rule of the operational semantics that can be applied to it.*

Proof For a detailed proof of this lemma, see [10]. □

Lemma 2 (Preservation) *If the model is well-typed and for an evaluation rule $\langle e, \sigma, \eta\rangle \rightarrow \langle \sigma', \eta'\rangle$ σ and η are well-typed so are σ' and η'. A model is well-typed, if the typing rules (see Sec. 3.1) can be applied without getting stuck. σ (and η respectively) is well-typed, iff for every binding $\sigma(p) = v$, v and p are of the same type, i. e., $p, v : T$.*

Proof For a detailed proof of this lemma, see [10]. □

Theorem 1 (Safety) *This type system is safe, i. e., there are no type errors at run-time.*

Proof Safety can be shown by proving progress and preservation [13]. □

Thus, when two components are compatible at compile time, they remain compatible during run-time. This is especially important for embedded systems, as a type error at runtime might lead to a damage of the system or its environment.

4 Measurement units

When a port in a model is assigned type Float, all we know is the range of values for that port. It does not say anything about what these values actually represent in the real world. A model in which two integers, say, one representing apples and the other oranges, are added would still be well typed. But in many cases it is not desirable that apples and oranges can be added. In such a case, additional semantic information must be represented in the model. This is achieved using *measurement units*.

Let $\mathbb{U}$ denote the set of all (measurement) units defined for a model. Which units are used within a model depends on the application domain. For embedded systems the SI units [12] could be used, as many physical values are processed in such an environment, i. e., $\mathbb{U} = \{\mathrm{s}, \mathrm{m}, \mathrm{kg}, \mathrm{A}, \mathrm{K}, \mathrm{mol}, \mathrm{cd}\}$. The function $\mu : \mathbb{U} \rightarrow \mathbb{Z}$ is a mapping from units $\mathbb{U}$ to integral numbers $\mathbb{Z}$ representing the exponent of the respective unit. This is necessary as mixed units, such as $acceleration = m/s^2$, can occur as well. This example would be represented as $\mu(m) = 1$, $\mu(s) = -2$ and $\mu(u) = 0$ for all remaining units $u \in \mathbb{U}$.

To be able to discern between different representations of the same unit, e. g., meters, inches, and miles, these can be encoded as separate units, all of the same dimension "length". It was just this mismatch between the metric and the imperial system of units that led to the loss of the Mars Climate Orbiter [7]. But a mismatch of the units of a model may not only be detected automatically, but in some cases even be resolved automatically, as some units can be converted into each other, e. g., meters and miles. For this automatic conversion, the function $\tau : \mathbb{U} \times \mathbb{U} \rightarrow component$ is introduced.

The component always has the same signature $i \rightarrowtail o$ with types $i, o : \forall X \sqsubseteq \mathsf{Float}.\ X$. Thus the conversion components can be type-checked with the same rules as the rest of the model. If two units cannot be converted, the function returns $\perp$.

The function τ can also be interpreted as a *conversion table*[1], as illustrated in an example with different units for the dimension length:

	m	cm	inch	mile
m	$\perp$	$x * 100$	$\perp$	$x/1609$
cm	$x/100$	$\perp$	$\perp$	$\perp$
inch	$\perp$	$\perp$	$\perp$	$x/63.360$
mile	$x * 1609$	$\perp$	$x * 63.360$	$\perp$

The conversion table τ does not have to be fully specified, since the conversion from cm to inch can be derived from the table above over m and mile. To achieve that, the conversion table can be interpreted as a *conversion graph* by creating a node for each table entry and connecting the nodes whenever the entry of the conversion table is not $\perp$. Thus the adjacency matrix A for this graph is given as:

$$A(\mu_1, \mu_2) = \begin{cases} 0 & \text{if } \tau(\mu_1, \mu_2) = \perp \\ 1 & \text{otherwise} \end{cases}$$

For the function τ to be consistent, it is required that, (1) if there is a path between any two units μ_1 and μ_2, there must also be a path between μ_2 and μ_1, (2) identical units are not converted, i. e., $\forall \mu.\ \tau(\mu, \mu) = \perp$ and (3) the conversion components do not contain any delays. Note that the adjacency matrix does not have to be symmetric to satisfy these constraints.

Lemma 3 *Two units μ_1, μ_2 are convertible if there exists a directed path in the adjacency matrix A from μ_1 to μ_2.*

Based on the adjacency matrix, the sets of convertible components could also be defined over the connected components of the conversion graph: Two units μ_1, μ_2 are convertible, if both belong to the same connected component of the graph defined by the adjacency matrix A.

The concept of units is not limited to the conversion based on scalar functions, as in [8], but may also be applied to arbitrary conversion functions, e. g., for Celsius, Kelvin and Fahrenheit:

[1] To make the table more concise, its components are represented as mathematical functions, rather than SCL components.

$$\frac{\Delta \vdash p_a, p_b, p_c : \mu}{\Delta \vdash plus(p_a, p_b \rightarrowtail p_c)} \quad \text{(D-plus)}$$

$$\frac{\Delta \vdash p_a : \mu_a \quad \Delta \vdash p_b : \mu_b \quad \Delta \vdash p_c : \mu_a \cdot \mu_b}{\Delta \vdash mult(p_a, p_b \rightarrowtail p_c)} \quad \text{(D-mult)}$$

$$\frac{\Delta \vdash p_a : \mu_a \quad \Delta \vdash p_b : \mu_b \quad \Delta \vdash p_c : \mu_a \cdot \mu_b^{-1}}{\Delta \vdash div(p_a, p_b \rightarrowtail p_c)} \quad \text{(D-div)}$$

$$\frac{\Delta \vdash p_a : \mu^2 \quad \Delta \vdash p_b : \mu}{\Delta \vdash sqrt(p_a \rightarrowtail p_b)} \quad \text{(D-sqrt)}$$

$$\frac{\Delta \vdash p_a, p_b : \mu}{\Delta \vdash eq(p_a, p_b \rightarrowtail p_c)} \quad \text{(D-eq)}$$

$$\frac{\Delta \vdash p_t, p_e, p_r : \mu}{\Delta \vdash ite(p_c, p_t, p_e \rightarrowtail p_r)} \quad \text{(D-if)}$$

$$\frac{\Delta \vdash p_a, p_b, v : \mu}{\Delta \vdash delay((p_a \rightarrowtail p_b), v)} \quad \text{(D-delay)}$$

$$\frac{\begin{array}{c}\forall c \in C.\ \Delta \vdash c \\ \forall c \in ch.\ \Delta \vdash c\end{array}}{\Delta \vdash component(S, ch, C, \Phi)} \quad \text{(D-net)}$$

$$\frac{\begin{array}{c}\Delta \vdash p_1 : \mu_1 \quad \Delta \vdash p_2 : \mu_2 \\ \mu_1 \neq \mu_2 \rightarrow convertible(\mu_1, \mu_2)\end{array}}{\Delta \vdash chan(p_1 \rightarrowtail p_2)} \quad \text{(D-chan)}$$

Figure 4. Unit inference rules of SCL

	C	F	K
C	$\bot$	$(x-32)/1,8$	$x-273$
F	$(x*1,8)+32$	$\bot$	$\bot$
K	$x+273$	$\bot$	$\bot$

Furthermore, the units are not restricted to physical units but may also be used, e. g., to convert between different currencies. Thus the choice of units depends largely on the application domain.

In contrast to [8] the units are treated separately from the type system. That is because units (in contrast to types) do not have an influence on the system behaviour at runtime. They are only used to check the consistency of the model. Thus we do not want to argue about them in the operational semantics and thus also in the safety proof.

The inference of units in a model is similar to the type inference. Thus the same notation is used as for the type system: In the unit context Δ port p has unit μ is written as $\Delta \vdash p : \mu$. Again for the different entities of an SCL model, different rules are required, the rules (T-port) and (T-perm) apply here as well. To be able to infer the units in a model, there are several operations required on μ. These are based on [8]. Let $n \in \mathbb{Z}$ in:

$$\begin{array}{lcl}\mu_1 = \mu_2 & \iff & \forall u \in \mathbb{U}.\, \mu_1(u) = \mu_2(u) \\ \mu_1 = \mu_2 \cdot \mu_3 & \iff & \forall u \in \mathbb{U}.\, \mu_1(u) = \mu_2(u) + \mu_3(u) \\ \mu_1 = {\mu_2}^n & \iff & \forall u \in \mathbb{U}.\, \mu_1(u) = n \cdot \mu_2(u) \\ \text{unitless}(\mu) & \iff & \forall u \in \mathbb{U}.\, \mu(u) = 0\end{array}$$

The test for equality $\mu_1 = \mu_2$ is required to check the consistency of the model. The addition of units $\mu_1 \cdot \mu_2$ is required for the mathematical multiplication and division operators, i. e., when two values are multiplied, their units are added. For the division operator units need to be inverted μ^{-1}. For the square root operator every unit will be divided by 2, i. e., μ^2. The predicate unitless(μ) is true if a port does not have any units.

The blocks in SCL are polymorphic in the sense that they can be used for any unit. There are only restrictions on the allowed combination of units. The rules are depicted in Fig. 4.

The $plus$ block only makes sense, if all ports have the same units. The rule for $minus$ is identical to (D-plus) modulo renaming. At the $mult$ block, the units have to be added, at the div block subtracted as defined in rules (D-mult) and (D-div). At the $sqrt$ block (D-sqrt), the exponents of the units of the input port must be divisible by two, since only integral numbers are allowed as exponents.

At the eq block, it does not make sense to compare values with different units (D-eq).

Analogue to the typing rules, a component is unit-consistent, if its channels and subunits are unit-consistent (D-net). The conversion of units is done at the channels, connecting ports with different units (D-chan). The function $convertible(\mu_1, \mu_2)$ is true, iff there exists a path from μ_1 to μ_2 in τ .

Since units only make sense on numerical types, all ports with units must have a subtype of Float. Thus for all ports with type $\Gamma \vdash p : T$ and unit $\Delta \vdash p : \mu$ the following property must hold:

$$\neg\text{unitless}(\mu) \rightarrow \Gamma \vdash T \sqsubseteq \mathsf{Float}$$

At compile time, after types and units have been checked and inferred successfully, this property is verified by the type checker. This provides the ability to use units not only on a single type, as in [8], but also on any subtype of Float.

The information about units is removed from the model at compile time, since it is no longer needed. The only thing that must remain in the model are components that convert between the different units. These components are generated automatically at compile time. Since the unit conversion only occurs at channels with different units, only these

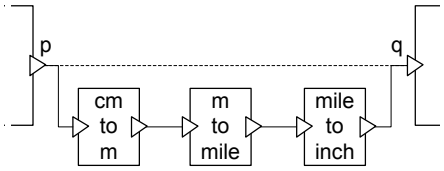

Figure 5. Conversion from cm to inch

channels need to be modified for unit conversion. These components are generated by replacing the original channel and linking the conversion components along the shortest path between cm and $inch$ in the conversion graph. The example in Fig. 5 illustrates the components inserted for this the conversion.

5 Conclusions

The framework of static compatibility presented in this paper comprises three different aspects: (1) data type checking, (2) unit checking, and (3) automatic data type and unit inference. This framework enables users to define syntactic interfaces of components by abstracting from concrete types and thereby allows a component to be (re-) used in several concrete contexts (i. e., polymorphic signatures). A concluding abstract example of type and unit inference is depicted in Fig. 6, assuming that the environment provides the appropriate types and units.

This general framework can easily be adopted to other component-based languages exhibiting different syntax and semantics, such as AutoFocus or MATLAB/Simulink, for instance. Notably, our notion of component compatibility can be checked statically, i. e., at compile-time, and therefore does not negatively affect runtime efficiency. Moreover, we have shown that this scheme does not leave space for subtle type-errors that may occur at runtime, and which are often experienced with standard component-based design and modelling tools that lack mechanisms of type inference and polymorphism.

In the present form, our approach can be realised with unification in only linear time w. r. t. the size of the model under scrutiny. However, if additional types, such as structured types (tuples, lists, etc.) are introduced to the model, the algorithm will be exponential in the worst-case. This is due to the fact that a component in a model could take as input, say, a tuple, and return, say, a tuple of tuples, and so forth.

Acknowledgements. The authors thank Stefan Berghofer for helpful discussions regarding the subtleties of type systems, and for his comments on draft versions of the paper.

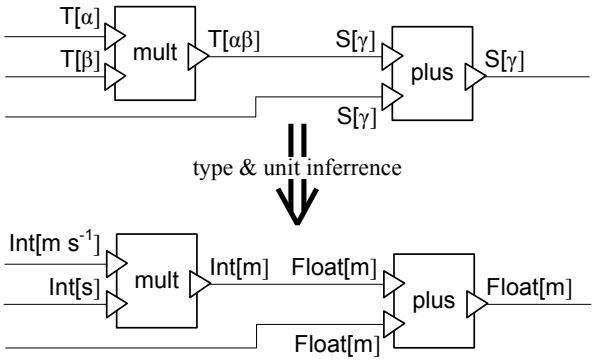

Figure 6. Type and unit inference

References

[1] ClawZ—The semantics of Simulink diagrams. Whitepaper, 2003. `www.lemma-one.com/clawz_docs`.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[3] G. Berry. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, chapter The Foundations of Esterel. MIT Press, 2000.

[4] M. Broy, F. Huber, and B. Schätz. AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. *Inform., Forsch. Entwickl.*, 14(3):121–134, 1999.

[5] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications. In *ACM SIGPLAN Conf. on Languages, compilers, and tools for embedded systems*. ACM Press, 2003.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. 8th European software engineering conference*, New York, NY, USA, 2001. ACM Press.

[7] E. E. Euler, S. D. Jolly, and H. H. Curtis. The failures of the mars climate orbiter and mars polar lander: A perspective from the people involved. In *Proc. Guidance and Control*. American Astronautical Society, 2001.

[8] A. J. Kennedy. Relational parametricity and units of measure. In *Proc. ACM SIGPLAN-SIGACT symposium on Principles of prog. lang.* ACM Press, 1997.

[9] K. Knight. Unification: a multidisciplinary survey. *ACM Comput. Surv.*, 21(1):93–124, 1989.

[10] C. Kühnel, A. Bauer, and M. Tautschnig. Compatibility and reuse in component-based systems via type and unit inference. Technical Report TUM-I0716, Technische Universität München, 2007.

[11] E. A. Lee and Y. Xiong. System-level types for component-based design. In *EMSOFT'01*. Springer, 2001.

[12] NIST. The NIST reference on Constants, Units and Uncertainty. `physics.nist.gov/cuu/Units`.

[13] B. C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[14] G. D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004.