Sound Transformations for Message Passing Systems*

P. Eugster
Department of Computer Science
Purdue University, USA
peugster@cs.purdue.edu

T. Freudenreich, S. Frischbier,
S. Appel, A. Buchmann
Department of Computer Science TU
Darmstadt, Germany
{lastname}@dvs.tu-darmstadt.de

ABSTRACT

Typing is a core issue in federated networked distributed applications, which are increasingly modeled as event-based systems. Constraining such a system to globally agreed types for exchanged event objects is impractical, and structural typing typically mediates only between types with attribute-wise correspondances.

This paper advocates a transformation-centric model for federated distributed applications. Our approach is generic in that it assumes the existence of certain global object types, which may be defined by a separate specification language and/or may arise during system deployment. Processes can define their individual sets of local types, and define conformance to global types via explicit transformations of values. Every process can thus be viewed as defining its own context, where transformations of incoming and outgoing objects can be manually programmed or generated.

We propose a flexible and expressive language to declare object transformations, covering the full spectrum: from symmetric equivalences between types to asymmetric projections of types or enrichment of objects which structural subtyping can not capture. We present a type system that resolves the relevant transformations and verifies that corresponding functions respect high-level mappings between global and local types. We present empirical evidence of the efficiency of our approach.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

Keywords

types, event, distributed, transformation, safety

*Funded in part by US NSF grants # 0644013 and # 0834529, DARPA grant # N11AP20014, Alexander von Humboldt foundation, and the German Federal Ministry of Education and Research (BMBF) grants # 01IA08006 and # 01IC10S01. Authors assume responsibility for content.

1. INTRODUCTION

Large-scale distributed software systems are the backbone of today's economy, enabling companies to cooperate across the globe. Corporate business generates a huge amount of information that has to be distributed effectively in nearly realtime among heterogeneous software components, and has to be interpreted correctly. This information often describes meaningful events in the form of event objects which are conveyed via network messages to all *subscribers* with matching interests. Communication happens across company and even national borders in an *n-to-m* fashion, often without direct references between communicating components.

The federated software systems supporting these communications are loosely-coupled, highly heterogeneous and developed by many parties. Thus we cannot assume that all information producers and consumers share the same interpretation of data (see Figure 1). Local interpretations — *contexts* — differ based on geographical, cultural, legal, or technical reasons. The issues faced here are similar to semantic data integration, but more stringent in terms of performance and flexibility: we must match data on the fly and event producers and consumers can join and leave dynamically.

Example applications.. As running example we adopt a scenario considered by two ongoing research projects — DynamoPLV (`www.dynamo-plv.de`) and EMERGENT (`www.software-cluster.org`) — investigating seamless integration of production, logistics, traffic management, and transportation.

Today's complex supply chains involve many companies worldwide, and production strategies like *just-in-time production* have strongly increased the need for continuous flows of information between participants in a chain. Manufacturers, for example, need to know the delivery status of key components, e.g., current position, to adapt as early as possible to disturbances in stock supply. However, there is a multitude of data formats to provide positioning information (e.g., latitude/longitude coordinates in GPS tracking systems) and almost every country has its own surface mail address format. Logistics providers, in turn, base their prices for transportation on a shipment's characteristics such as weight, dimensions, fragility, or risk of deterioration. Moreover, shipments with high priority (e.g., guaranteed delivery within a given timeframe) are usually delivered using different means than low priority shipments. All this information has to be considered by logistics providers when trying to trade off optimal resource utilization while meeting individual delivery conditions. Thus, these providers have to collect information about shipments in advance and continuously monitor these shipments and traffic situations along delivery routes. In world-wide settings, though, companies use different unit systems (e.g., Metric vs. Imperial); even within one unit system, there are alternatives (e.g., grams vs. kilograms).

Scenarios like the one sketched out here will be more common given the continuing globalization of economy; the vision of "ubiquitous computing resources" promoted by the cloud paradigm and the availability of event notification systems for clouds (e.g., Amazon Simple Notification Service [1]) will technically support the trend. Similarly, the proliferation of social networks interconnecting people with different cultural backgrounds and the use of notification services for communication therein (e.g., LinkedIn's Kafka [2]) will increase the need for mediation between data types.

Federated event objects..

Existing approaches to typing of data in distributed message passing systems falls broadly into the well-known approaches of *nominal* or *structural* typing. As an example of the former approach, Java RMI extends Java's static nominal typing to method invocations on remote objects. Just like several research languages or language extensions for distributed programming [13, 14], most work on *n-to-m* dispatching of events (e.g., [7, 18]) promotes the latter approach, following the model of *self-describing messages* [16] – maps of key-value pairs which are manually populated by producers and inspected by consumers.

Both approaches have well-known benefits. Nominal typing supports *efficiency*, as decentralized dispatching of event objects based on type identifiers is more efficient than full traversals of objects. Static nominal typing can catch many errors at compilation, improving *safety*. Structural typing inversely can lead to accidental matching. Nominal typing, however, forces an agreement, hampering *flexibility* broadly. Legacy system integration requires changes to independently developed components. Furthermore, providing adaptability of deployed types can require tediously coordinated updates. Interoperability is another issue: even within a language, two sets of modules developed independently are not always easily integrated. For instance, single inheritance may keep a class from being adapted to a framework.

In the applications mentioned in Section 1 both nominal and structural typing fall short. Nominal typing is an intuitive first choice, as most such applications are written in languages like Java, C++, or C#. Changes with respect to event object types, when they occur though, are not well accommodated, and components written in different languages can hardly interact. Structural typing provides such flexibility, but only mediates between types and not *values*. That is, several relevant scenarios remain unaddressed:

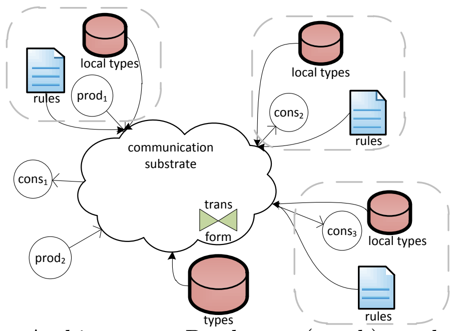


Figure 1: Architecture. Producers ($prod_i$) send event notifications to consumers ($cons_j$) through a communication substrate. A client supplies mappings and transformation rules to mediate between the global types and its local types (omitted for $cons_1$, $prod_2$).

- *automatic changes of attribute values*: the case of translating between units, e.g., Fahrenheit and Celsius.
- *enrichment*: adding a field with a default or **null** value, e.g., the state for a European surface mail address in US format.
- *complex conversions*: converting entities encoded with several primitive values, without 1-1 correspondence of such attributes between types, e.g., Metric and Imperial system, Cartesian (abscissa/ordinate) and polar coordinates (radius/azimuth).

Our target applications require such higher *expressivity*. Note that other approaches to interopability in distributed systems (e.g., OMG's CORBA) focus on mediating between *encodings* of values in a same unit (little endian vs big endian).

A transformation approach..

The approach proposed in this paper consists in *embracing* the differences between types, and generalizing these to a model centered on *transformations* (see Figure 1). That is, we assume a set of global event types, which can be defined through a dedicated specification language with mappings to different concrete programming languages, or can simply be the first types to have been used in a growing application.

Every process can be in a distinct context with regards to the types it considers. We introduce an explicit notion of *context* for processes, including (1) local types, (2) transformation rules to specify the desired fine-grained transformations to these local types from global ones and vice-versa, and (3) high-level mappings used to verify that the rule application yields consistent outcomes. Being in a message passing model where processes operate on respective copies of notifications, our transformations need not be bi-directional as in predating work on shared data [11].

Contributions and roadmap..

More specifically, in this paper we

1. introduce our rule-based transformation approach via

concrete examples, arguing for *flexibility* and *expressivity* (Section 2).

2. present a core language modeling the targeted event-based architectures and supporting transformations (Section 3).

3. propose a type system for our language achieving *safety* by pre-selecting and verifying transformation rules (Section 4) which are applied on nested event objects at runtime (Section 5); we extend our model to deal with recursive types (Section 6).

4. evaluate an implementation of our approach for Java based on ActiveMQ [21], demonstrating its *efficiency* (Section 7).

Section 8 discusses related work. Section 9 draws conclusions. Additional material and benchmarks can be found at `http://www.dvs.tu-darmstadt.de/research/events/actress/`.

2. DESIGN OVERVIEW

We provide a stepwise introduction into event object transformations through examples from the scenario introduced in Section 1. Figure 2 outlines types and transformations used for illustration.

2.1 Preliminaries: Event Objects and Transformations

We consider in this paper event objects in the form of typed records τ[...] of attributes; in a nested fashion, such attributes can be objects with attributes. For example, InvoiceLine [...] represents an object of type InvoiceLine defined in Figure 2. The record [...] contains a sequence of objects corresponding to the attributes of InvoiceLine (i.e., amountToPay, price, spec) which are of respective types defined by InvoiceLine (Money, Money, and ItemSpecification).

A very simple form of transformation of such objects consists in modifying values of attributes which are of specific primitive types such as **float**s. These attributes can have a unit associated with them, e.g., cm or inch (cf. InvoiceLine.amountToPay). A similar case are conversions of primitive values between different architectures or platforms. On the other end of the spectrum, an event may be transformed in a way which affects its internal structure, for example by merging multiple attributes, dropping attributes, and instantiating new ones. Any combination of these may be used to deal with versioning – by adding version numbers to type names and describing corresponding transformations, large scale systems can be updated to new type versions without interrupting functionality.

We now introduce a model of transformations covering the whole spectrum. Our model is centered around *transformation rules* — *t*-rules for short — which minimize the necessary specifications and align well with the mental model of programmers.

2.2 Separating Rules and Functions

The logistics provider from our example in Section 1 receives the specifications for the items to be transported from its customers. The caveat with this approach is that specifications do not have a standard format and even simple things like units vary between countries or even individual customers.

Intuitively, we would like the ability to define "default" transformations for certain types. Suppose a software service that calculates the price for transporting goods. As part of the calculation, it processes instances of ItemSpecification which contain information in a specific combination of units (e.g. meters for height, width and depth). After the calculation is complete, the logistics provider sends the price along with these specifications to its customers. A US customer is used to getting these properties in Imperial units (e.g. inches) rather than the logistics provider's internal format. A t-rule to transform all such attributes in all event objects could be simply expressed as (cf. r_1 in Figure 2):

```
ItemSpecification ▷ toUSSpecification;
```

The syntax of t-rules such as the above is roughly of the form $b \rhd f$ where b is a *pattern* delineating a set of attributes in event type(s) and f refers to a function (or method in an object-oriented language). This function is defined separately and used to transform such attributes. In the example above toUSSpecification refers to a function which takes an instance of ItemSpecification as its argument and produces an instance of an analogous type USItemSpecification, which is local to the present context:

```
USItemSpecification toUSSpecification( ItemSpecification is)
  { return new USItemSpecification (...); }
```

2.3 Nesting Level

For convenience we let a pattern like the above apply *at any nesting level* within a type. Specifically, the pattern applies to any attribute of type ItemSpecification at any depth in events of any type.

There are cases where we want to leave certain attributes in *specific* event types unchanged (or apply a different transformation function). For example, customs declaration papers require the same units as the logistics provider usually uses. In this case, the logistics provider does not want to transform the item specification, but just use it as it is. To achieve this, we can qualify more precisely where to apply this sort of transformation. The following t-rule

```
CustomsDeclaration. ItemSpecification ▷ toIdentity;
```

would apply an identity function (omitted here for brevity) to attributes of type ItemSpecification in CustomsDeclarations.

To overcome the cumbersome task of enumerating all event types which contain ItemSpecifications, we introduce intuitive priorities among rules. If we combine both t-rules above into one t-rule set, the second t-rule overrides the first one for CustomsDeclaration events. All other attributes of type ItemSpecification are transformed according to t-rule r_1 as depicted in Figure 2. This conveys the first intuition underlying our design: rules with more specific patterns override rule with less specific ones. In the above example, CustomsDeclaration.ItemSpecification overrides ItemSpecification.

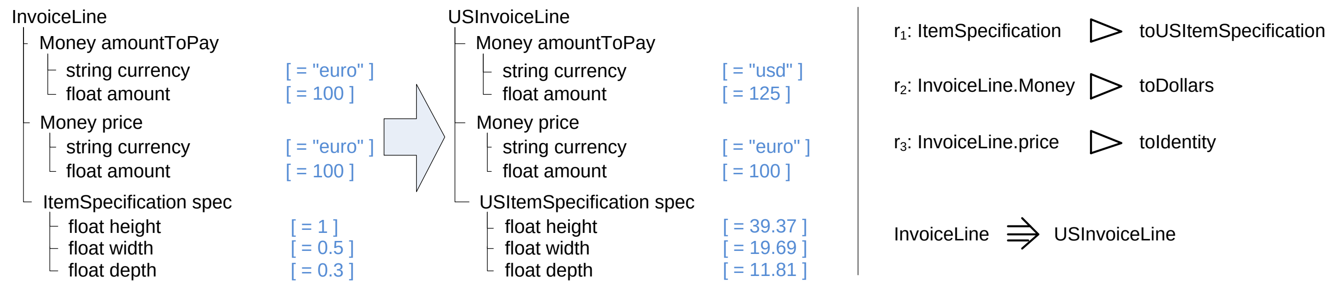


Figure 2: Sample transformation scenario. Simplified transformation rules (t-rules) and mappings are illustrated on the right.

Specificity here translates to length, or, in other terms, *nesting level*.

As mentioned we consider patterns ending in types to apply to all occurrences of that type in deeper nesting levels. That is, one can picture InvoiceLine.Money as representing InvoiceLine.∗Money where ∗ can match any (possibly empty) infix. This seems natural when looking at the generic transformations specified in the preceding examples. We apply this variable nesting level only at the *last* type in a pattern, prohibiting something like TopLevelType.∗LevelX.∗LevelY. More expressive patterns could be envisioned by removing this constraint but this is likely to come at a substantial cost in terms of simplicity for the programmer.

2.4 Instances over Types

Transforming by type only is sometimes too general. Consider the type InvoiceLine of Figure 2, which represents a single line on an invoice and contains the item's specifications and the cost for delivery (with its own Money type). Assume that the logistics provider needs this amount in its currency for internal bookkeeping. However, customers want this amount in their local currencies. Thus we cannot treat every attribute of type Money the same way.

We consequently support references to *attributes* as another intuitive element in patterns for t-rule application. The t-rule

```
InvoiceLine.amountToPay ▷ toDollars;
```

applies the following function

```
Money toDollars(Money m) {
  if (m.getCurrency() == "euro") {
    m.setCurrency("usd");
    m.setAmount(m.getAmount() * 1.25);
  } else if (m.getCurrency() == "yen") {...}
  ...
  return m;
}
```

to only the amountToPay *attribute* of events of the InvoiceLine type. Other attributes of type Money in an InvoiceLine or any other type are left unchanged (cf. Figure 2). Even if *succeeded* by a less specific t-rule which mandates that all Money attributes be transformed to include taxes, such as in the t-rule set

```
InvoiceLine.amountToPay ▷ toDollars;
Money ▷ addTaxToMoney;
```

the attribute amountToPay would be transformed via toDollars and not addTaxToMoney.

We believe that this is a much more natural semantics than declaration order alone. The intuition behind this second form of precedence consists in prioritizing attributes over types on otherwise comparable patterns, i.e., instance-level declarations over type-level declarations. The outcome above would be identical if the second t-rule used the pattern InvoiceLine.Money to define a default translation of all money attributes in type InvoiceLine.

2.5 Subtyping Level

Subtyping is a fundamental concept in programming languages. With nominal subtyping in our model, any event object can at any nesting level have an attribute a carrying an instance of a type τ' which is a subtype of a's declared type τ. With that in mind, it seems natural to allow the programmer to define t-rules which are subtype-sensitive. For instance, we might define a type WeightedItemSpec which extends ItemSpec by adding an attribute weight of type **float**, and define a corresponding *specific* rule

```
InvoiceLine.spec(WeightedItemSpec) ▷ ...;
```

It seems natural to favor among a set of competing patterns the one which refers to the most *derived* type at a point of comparison, e.g., selecting the above rule over one with pattern InvoiceLine.spec (defaulting to InvoiceLine.spec(ItemSpecification)) for an attribute weight of *dynamic* type WeightedItemSpec in an InvoiceLine. This selection is similar to (single) dynamic method dispatching.

2.6 Discussion

The three natural intuitions on precedence (nesting level, instances over types, subtyping level) can conflict. Among two patterns b_1 and b_2, b_1 may exhibit a deeper nesting level but end in a type, while b_2 ends in an attribute qualifier; similarly, b_1 may end in an attribute a and b_2 in a type τ, with a of a declared type τ which is a super-type of τ'. This demonstrates the need to precisely define the semantics of patterns and t-rules altogether.

Similarly, when defining t-rules based on patterns for transforming arbitrarily nested attributes, it becomes necessary to ensure that the transformations respect *mappings* of types at the highest level which for instance state that incoming events of a type τ are mapped to a type τ' ($\tau \Rrightarrow \tau'$). Some of the above transformation functions yielded types differing from their arguments, and thus return values can not be

$$
\begin{array}{lll}
\textit{program} & \Pi & ::= \emptyset \mid \Pi \parallel g \mid \Pi \parallel p \mid \Pi \parallel m \\
\textit{incarnation} & p & ::= P\langle \overline{e}, \overline{e} \rangle \\
\textit{subscription} & s & ::= \textbf{subscribe}(\tau\ x)\{e\} \\
\textit{expression} & e & ::= \textbf{publish}(e) \mid f(\overline{e}) \mid \tau[\overline{e}] \mid x \mid e.a \mid v \\
\textit{value} & v & ::= m \\
\textit{notification} & m & ::= \tau[\overline{v}] \\
\textit{context} & c & ::= (\overline{d}, \overline{u}, \overline{r}) \\
\textit{global} & g & ::= \gamma\ \textbf{ext}\ \gamma[\overline{\tau}\ \overline{a}] \\
\textit{local} & l & ::= \sigma\ \textbf{ext}\ \tau[\overline{\tau}\ \overline{a}] \\
\textit{type} & \tau & ::= \gamma \mid \sigma \\
\textit{function} & d & ::= \tau\ f(\overline{\tau}\ \ \overline{x})\{e\} \\
\textit{pattern} & b & ::= \tau \mid b.q \\
\textit{qualifier} & q & ::= a(\tau) \mid \tau \\
\textit{mapping} & u & ::= \tau \Rrightarrow^{h} \tau \\
\textit{t-rule} & r & ::= b \rhd^{h} f \\
\textit{direction} & h & ::= ? \mid !
\end{array}
$$

Figure 3: Syntax

simply plugged as substitutes for the original ones; the types in the entire path that led down to that attribute must be adapted. For instance, when substituting an ItemSpecification by a USItemSpecification the parent type has to change, e.g., to USInvoiceLine (see Figure 2). We proceed in the following with formalizing the intuitions behind our transformations and mappings.

3. A CORE LANGUAGE

We introduce a core programming language incorporating support for transformatons. Our language introduces processes which bear superficial similarities with Actors [5] and are reminiscent of process algebras like π-Calculus [19]. However our language uses *multi*cast communication and incorporates static typing, yet makes simplifications elsewhere, e.g., omitting higher-order processes.

3.1 Syntax

$\overline{z}$ denotes a sequence of several definitions z, independently of separating symbols (e.g., commas in $z_1, z_2, \ldots$). $\overline{z}$ is numbered z_1 to z_n by default, with $n = 0$ indicating an empty sequence; the only exception are sequences of type qualifiers $\overline{\tau}$ in patterns of t-rules which are numbered starting at τ_0 because any pattern contains at least one type qualifier. When constructing a sequence from a set (e.g., by assignment $\overline{z}=\{\ldots\}$) any order may be used.

Processes, types, and expressions.. We consider programs Π consisting in multiple distributed interacting processes. A process *declaration* is of the form

$$P(\overline{l}, c, \overline{s}, \overline{e})$$

where P is the name of the process; $\overline{l}$ refers to a set of local type declarations, c is a context which describes transformations, $\overline{s}$ is a set of handlers of the form **subscribe**$(\tau\ x)\{\ldots\}$ ("subscriptions") to react to τ events, and $\overline{e}$ is P's body. For example **subscribe**(USItemSpecification s){…} is a subscription to USItemSpecifications introduced in Section 2 (see Figure 2).

Figure 3 presents these constituents of processes and our syntax in more detail. A running program Π consists in a parallel composition of (a) a set of global types $\overline{g}$, (b) a set of process *incarnations* $\overline{p}$, each of the form $P\langle \overline{e}, \overline{e}' \rangle$ (current body $\overline{e}$, outgoing notifications under transformation $\overline{e}'$), and (c) a set of in-transit event notifications $\overline{m}$. As usual parallel composition $\parallel$ is associative and commutative. A process incarnation $P\langle \ldots, \ldots \rangle$ is thus the runtime representation of a process declared as $P(\ldots, \ldots, \ldots, \ldots)$.

A type τ defines attributes with respective types. All types have a super-type, at the exception of the abstract root type $\perp$. As usual in practice, if left unspecified in a type declaration, the ancestor of a type becomes automatically $\perp$. The Money type introduced in Section 2 (see Figure 2) for example is represented as Money[**string** currency, **float** amount]. For simplicity, as common, we refrain from detailing primitive types and values and their manipulation in our formalism. We prohibit (transitively) recursive type definitions at this point (we handle these in Section 6), and omit member functions/methods from types. An event is instantiated as a nested expression; simply put, an event is an object that is published (multicast). We use the term *notification* to refer specifically to such an expression which is a value (normal form).

Expressions include, as mentioned, nested objects of which event notifications m are only a special case; these last ones are simply nested typed records of values $(\tau[\overline{v}])$. Expressions furthermore include publications **publish**(e) and function calls $f(\overline{e})$. We omit more complex expressions like while loops typical of Actors (for describing bodies of processes and reactions), or *let* expressions, to not distract from the main issues of expressing contexts in a flexible and safe manner.

Contexts, t-rules, mappings, and patterns.. A context description c includes a set of definitions of functions f which are typically used for transforming notifications (e.g., toUSItemSpecification in Figure 2). In our Java prototype implementation described later in Section 7.1, such functions are methods and can obtain information on the path (e.g., InvoiceLine.price in Figure 2) at which transformation is currently taking place. Since we assume at most one incarnation for a given process declaration there is a 1-1 correspondance between processes and contexts, and thus we henceforth may refer to a process as (defining) a context.

A context further states a set of (unidirectional) mappings u from global types to possibly local, context-dependent, types and vice-versa. $\tau \Rrightarrow^{!} \gamma$ represents a mapping from (possibly local) type τ to a global type γ, where τ is a type of events *published* by the respective process; inversely, $\gamma \Rrightarrow^{?} \tau$ represents a mapping to a *subscribed* event type τ from a global type γ. The label h thus distinguishes between transformations for incoming (h=?) and outgoing (h=!) notifications. A complete mapping for the example of Section 2 would be InvoiceLine $\Rrightarrow^{?}$ USInvoiceLine (see Figure 2).

A process can directly subscribe to (or publish instances of) global types of course and the corresponding mapping is trivial and can be omitted in a concrete language but is assumed here nonetheless for streamlining definitions. Note that similarly, for presentation simplicity, a process may only have a single mapping and a single subscription for a given global event type.

While mappings define at a high level *which* local and global types correspond to each other, a context further contains a set of more fine-grained *t-rules* defining *how* to transform instances of (possibly) local to global types and vice-versa. A t-rule consists in a pattern b and a reference to a function f (e.g., r_1-r_3 in Figure 2). A t-rule applies either to published notifications ($b \rhd^{!} f$) or to received notifications ($b \rhd^{?} f$). The first example in Section 2.2 would thus be fully encoded as $\mathsf{ItemSpecification} \rhd^{?} \mathsf{toUSSpecification}$. A pattern b is a sequence of qualifiers q each of which, at the exception of the first, is either an attribute with a type name or simply a type name: (1) An *attribute qualifier* $a(\tau)$ in $b.a(\tau)$ denotes the (nested) attribute a of type τ at the path expressed through its prefix b in a notification of the type captured by the first qualifier in b. The type τ in $a(\tau)$ is used for increasing expressiveness in the face of subtyping; omitting it in practice by writing simply $b.a$ means that the static type τ of a according to b is selected, for example, τ_1 for a_1 in a pattern $\tau_0.a_1$ with τ_0 $\mathbf{ext}$ ...[...τ_1 a_1...]. Subtyping is supported in that we can describe a (additional) t-rule with a pattern $\tau_0.a(\tau_1')$ where τ_1' is a subtype of τ_1. (2) A *type qualifier* τ at the first or other position in a pattern refers to *all attributes* of the corresponding type at the path expressed through its prefix; if it occurs at the end of a pattern it also applies to *any deeper nestings from there*. Thus $\tau_0.\tau_1$ denotes all attributes of type τ_1 (or subtypes) in notifications of type τ_0 or, recursively, in any other attributes defined by τ. The pattern $\tau_0.a.\tau_2$ denotes all (nested) attributes of type τ_2 in attribute a in event type τ_0. As with attribute qualifiers, type qualifiers support subtyping. For instance, we can define a t-rule with a pattern $\tau_0.\tau_1$ where there is no (nested) attribute of the exact type τ_1 in τ_0; it suffices that there is an attribute of a super-type τ_1' of τ_1. For simplicity we assume that no two t-rules can use identical patterns. In practice we can deterministically chose, e.g., the last one among such competing t-rules and issue a warning.

3.2 Auxiliary Functions and Subtyping

We define in Figure 4 some auxiliary functions that allow us to inquire about characteristics of types and processes at compilation and at runtime. Any function is sensitive to the context in which it is evaluated, and thus carries the corresponding process identifier P as subscript. A function evaluates to $\bullet$ for a given set of arguments if the function is not defined for that set. $attrs_P(\tau)$ yields the signature of a given type τ in context P. $ftype_P(f)$ provides the signature of a function f defined in a given context P. $fbody_P(f)$ yields the formal arguments and body of function f. $subs_P$ returns all subscribed types of P. $react_P(\tau)$ returns the formal argument and body of P's reaction to events of type τ; if P does not define a reaction for τ then the reaction for its *immediate* super-type ($\prec^1_P$, see Figure 5) τ' is chosen. $rule^h_P(b)$ yields the function associated with the pattern b for h transformations in context P. $map^h_P(\tau) = \tau'$ denotes that incoming (h=?) or outgoing (h=!) notifications of type τ are mapped to notifications of type τ'. If there is no specific mapping for a given type τ, its mapping is that of its ancestor ([M-Type-Inh]).

$$\frac{P(\ldots, c, \ldots, \ldots) \quad c = (\ldots, \overline{u}\ \tau \Rrightarrow^{h} \tau\ \overline{u}', \ldots)}{map^h_P(\tau) = \tau}\ \text{[M-Type]}$$

$$\frac{\tau \prec^1_P \tau' \quad P(\ldots, c, \ldots, \ldots) \quad c = (\ldots, \overline{u}, \ldots) \quad \nexists\, \tau \Rrightarrow^{h} \tau \in \overline{u}}{map^h_P(\tau) = map^h_P(\tau')}\ \text{[M-Type-Inh]}$$

$$\frac{\tau \prec^1_P \tau'\ attrs_P(\tau') = \langle \overline{a}', \overline{\tau}' \rangle}{attrs_P(\tau) = \langle \overline{a}'\overline{a}, \overline{\tau}'\overline{\tau} \rangle}\ \text{[E-Type]}$$

$$attrs_P(\bot) = \langle \emptyset, \emptyset \rangle\ \text{[A-Type-B]}$$

$$map^h_P(\bot) = \bullet\ \text{[M-Type-B]}$$

$$\frac{P(\ldots, c, \ldots, \ldots) \quad c = (\overline{d}\ \tau\ f(\overline{\tau}\ \overline{x})\{e\}\ \overline{d}', \ldots, \ldots)}{ftype_P(f) = \overline{\tau} \rightarrow \tau}\ \text{[F-Type]}$$

$$\frac{P(\ldots, c, \ldots, \ldots) \quad c = (\overline{d}\ \tau\ f(\overline{\tau}\ \overline{x})\{e\}\ \overline{d}', \ldots, \ldots)}{fbody_P(f) = \langle \overline{x}, e \rangle}\ \text{[F-Body]}$$

$$\frac{P(\ldots, \ldots, \overline{s}, \ldots) \quad \overline{s} = \overline{s}'\ \mathbf{subscribe}(\tau\ x)\{e\}\ \overline{s}''}{react_P(\tau) = \langle x, e \rangle}\ \text{[R-Body]}$$

$$\frac{\tau \prec^1_P \tau'\ P(\ldots, \ldots, \overline{s}, \ldots) \quad \nexists \mathbf{subscribe}(\tau\ x)\{\ldots\} \in \overline{s}}{react_P(\tau) = react_P(\tau')}\ \text{[R-Body-Inh]}$$

$$\frac{P(\ldots, c, \ldots, \ldots) \quad r = b \rhd^{h} f \quad c = (\ldots, \ldots, \overline{r}\ r\ \overline{r}')}{rule^h_P(b) = f}\ \text{[P-Fun]}$$

$$\frac{P(\ldots, c, \ldots, \ldots) \quad c = (\ldots, \overline{u}, \ldots) \quad \overline{\gamma} = \{\gamma \mid \gamma \Rrightarrow^{?} \tau \in \overline{u}\}}{subs_P = \overline{\gamma}}\ \text{[Sub-Types]}$$

Figure 4: Auxiliary functions. Subtyping $\prec^1$ is defined in Figure 5

We introduce two subtyping relations. $\tau \preceq_P \tau'$, as common, states that τ is a subtype of τ' and $\preceq$ is a reflexive and transitive relation. $\tau \prec^1_P \tau'$ states that τ is a *direct* descendent of τ', which means τ is explicitly declared to be a subtype of τ' (τ $\mathbf{ext}$ τ'[...]), which can happen either in global or local type definitions; $\prec^1$ is thus neither transitive nor reflexive. For more compact notation $\preceq$ is defined in terms of $\prec^1$ ([S-Gen]) by adding transitivity ([S-Trans]) and reflexivity ([S-Refl]).

$$\frac{\Pi = \gamma\ \mathbf{ext}\ \gamma'[\ldots] \parallel \ldots}{\gamma \prec^1_P \gamma'}\ \text{[S-GDef]}$$

$$\frac{P\langle \bar{l}\ \sigma\ \mathbf{ext}\ \tau[\ldots]\ \bar{l}', \ldots, \ldots, \ldots \rangle}{\sigma \prec^1_P \tau}\ \text{[S-LDef]}$$

$$\tau \preceq_P \tau\ \text{[S-Refl]}$$

$$\frac{\tau \preceq_P \tau' \quad \tau' \preceq_P \tau''}{\tau \preceq_P \tau''}\ \text{[S-Trans]}$$

$$\frac{\tau \prec^1_P \tau'}{\tau \preceq_P \tau'}\ \text{[S-Gen]}$$

Figure 5: Subtyping relations $\tau \prec^1_P \tau$ and $\tau \preceq_P \tau$

4. TRANSFORMATION RESOLUTION AND VERIFICATION

We now proceed to defining our transformation semantics, and formally characterize it via a type system.

4.1 Overview

Type checking of any individual process P prior to its deployment ensures that P correctly transforms any incoming and outgoing notifications such as to abide to the global event types and its own mappings. In the following we outline the intuitions behind selection of t-rules and conformance checking.

Paths.. Our approach verifies for any *fully qualified path*, defined below, any transformation functions f to be applied at that path, and verifies whether the type returned by f abides to the type stipulated by the mapping for the corresponding event type.

Definition 1 (Fully qualified path). *A fully qualified path* $\langle \tau_0 ... \tau_n, a_1 ... a_n \rangle$ *for process* P *is such that* $\exists \tau'_1 ... \tau'_n, \forall i \in [0..n-1], attrs_P(\tau_i) = \langle ... a_{i+1} ..., ... \tau'_{i+1} ... \rangle$ *and* $\tau_i \preceq_P \tau'_i$.

A fully qualified path thus unambiguously and correctly denotes an object within a notification. In the following whenever referring to paths we mean such fully qualified paths. A prefix of a path (e.g., $\langle \tau_0 \tau_1, a_1 \rangle$ for $\langle \tau_0 \tau_1 \tau_2, a_1 a_2 \rangle$) is a path itself.

For every process P our type system identifies for any given event type τ mapped in or out by P all transformations for all *reachable* paths rooted at τ, and retains these. This retained information is of the form $\langle \tau_0 ... \tau_n, a_1 ... a_n, f \rangle$, prompting the evaluation semantics to apply function f at the path $\langle \tau_0 ... \tau_n, a_1 ... a_n \rangle$ in any event of type τ_0. These t-rules are resolved by starting from all subscribed and published types τ_0, and exploring their attribute spaces recursively by following *breadth* first (e.g., $\forall a_1$ s.t. a_1 is declared by τ_0) and then *width* (e.g., $\forall$ a_2 s.t. $a'_1 s$ type τ_1 declares an attribute a_2). To deal with subtyping, for a given path (e.g., $\langle \tau_0 ... \tau_i, a_1 ... a_i \rangle$) the *subtype space* is explored similarly in a recursive manner (e.g., $\forall \tau'_i \prec^1_P \tau_i$). A reachable path implies that there is no transformation for any of its prefixes; nested exploration does not proceed further when a transformation is identified, as the respective function is responsible for dealing with nested attributes.

This prevents us from having to resolve any t-rules at runtime. Upon addition of new types at runtime, individual processes can re-run the type checking and t-rule resolution; in practice this can be done in an incremental fashion.

Relaxed conformance.. Remember that there is not necessarily a 1-1 relationship between mappings and t-rules; in fact that would be undesirable in terms of expressiveness. A mapping can involve the application of multiple t-rules, and inversely, a t-rule may be applied by different mappings.

We furthermore strive for a minimal conformance verification. More precisely, we do not mandate that every t-rule in a context respects all mappings for event types with paths matching the t-rule's pattern, including those which that t-rule is never applied to due to priorities among patterns. This allows for default t-rules which typically include type qualifiers in their patterns to be overridden by more attribute-specific t-rules. The latter ones produce the correct type at a given path but the former ones — if applied there instead — would not necessarily do so.

$$type(a(\tau)) = \tau \;\text{[Q-Type-A]} \qquad type(\tau) = \tau \;\text{[Q-Type-T]} \qquad a(\tau) \trianglelefteq_P \tau \;\text{[Inst-Prio]}$$

$$\frac{type(q) \preceq_P type(q')}{q \trianglelefteq_P q'} \;\text{[Subtype-Prio]} \qquad \frac{b \trianglelefteq_P b' \quad q \trianglelefteq_P q'}{b.q \trianglelefteq_P b'.q'} \;\text{[Nested-Prio]} \qquad \frac{b \trianglelefteq_P b'}{b.q \trianglelefteq_P b'} \;\text{[Depth-Prio]}$$

Figure 6: Priorities among patterns $b \trianglelefteq_P b'$

Priorities.. We use the following priorities for competing t-rules:

P1. Nesting level: A natural choice consists in considering nesting level as prioritizing measure among otherwise equivalent patterns (see Section 2.3). Deeper nesting levels translate to more detailed knowledge about data-structures and thus to more specific behavior. Thus a t-rule with the pattern $\tau_0.\tau_1$ will be chosen over one with pattern τ_1 for attributes of type τ_1 in notifications of type τ_0.

P2. Instances over types: Another natural choice (see Section 2.4) consists in giving attribute qualifiers priority over type qualifiers among otherwise equivalent patterns. Thus $\tau_0.a_1(\tau_1)$ would be chosen over $\tau_0.\tau_1$ for attribute a_1 in a notification of type τ_0.

P3. Subtyping level: A third natural choice (see Section 2.5) is to consider for any otherwise equivalent patterns the ones which use qualifiers with the most derived types. Thus, with τ'_1 being a strict subtype of τ_1, $\tau_0.a_1(\tau'_1)$ is chosen over $\tau_0.a_1(\tau_1)$.

P4. Instances over nesting level: We need to break ties between P1 and P2 (see Section 2.6). Assume we are transforming at a path $\langle \tau_0 ... \tau_3, a_1 a_2 a_3 \rangle$. Now consider two matching patterns (a) $\tau_0.a_1(\tau_1).\tau_3$ and (b) $\tau_0.\tau_1.\tau_2.\tau_3$. Clearly, (a) is more specific than (b) according to P2, but (b) has a deeper nesting level than (a) which thus far prevails according to P1. Even if there are no attributes of type τ_3 immediately in a_1, τ_3 is expressed for that prefix $\tau_0.a_1(\tau_1)$ which is more specific than the corresponding prefix $\tau_0.\tau_1$ in pattern (b), and thus (a) is prioritized.

P5. Subtypes over instances: Finally, we need to break ties between P3 and P1, and between P3 and P2. Both of these ties are broken by favoring subtyping over instances (P3 over P4). That is, we give priority to the type of a qualifier rather than whether the qualifier refers to the considered attribute or only its type; between a type qualifier τ and an attribute qualifier $a(\tau)$ in the same position with the same type τ we follow P2.

These priorities are captured by the relation $b \trianglelefteq b'$ defined by the rules in Figure 6. The *type*(q) helper function simply returns the type of an attribute qualifier ([Type-A]) or type qualifer ([Type-T]). Note that the t-rule resolution semantics presented shortly can be used with other priorities.

$$\frac{\models_P b.a(\tau) : \tau}{\models^{\diamond}_P b.a(\tau) : \tau} \;\text{[Qual-Attr-T]} \qquad \models^{\diamond}_P \tau : \tau \;\text{[Qual-T]} \qquad \models_P \tau : \tau \;\text{[Qual-Consec-T]}$$

$$\frac{\models_P b : \tau_0 \qquad \exists a_1 ... a_n, \tau_1 ... \tau_n \mid \forall\, i [\in 1..n]\, attrs_P(\tau_{i-1}) = \langle ... a_i ..., ... \tau_i ... \rangle \wedge \tau \preceq_P \tau_n}{\models^{\diamond}_P b.\tau : \tau} \;\text{[Qual-Type-T]}$$

$$\frac{\models_P b : \tau' \quad \tau \preceq_P \tau_i \quad attrs_P(\tau') = \langle ... a_i ..., ... \tau_i ... \rangle}{\models_P b.a_i(\tau) : \tau} \;\text{[Qual-Attr-Consec-T]} \qquad \frac{\models_P b : \tau' \quad \tau_i \preceq_P \tau \quad attrs_P(\tau') = \langle ... a_i ..., ... \tau_i ... \rangle}{\models_P b.\tau : \tau} \;\text{[Qual-Type-Consec-T]}$$

Figure 7: Typing judgements for patterns – $\models_P b : \tau$ and $\models^{\diamond}_P b : \tau$

4.2 Pattern Typing and Applicability

Next we describe how to validate patterns, e.g., verify whether for a pattern $\tau_0.a_1(\tau_1)$ type τ_0 actually does contain an attribute a_1 of τ_1 or of a super-type of τ_1. We are conservative here, meaning that the *possibility* of some subtype $\tau_0' \preceq_P \tau_0$ containing some a_1 but not τ_0 does not suffice; a pattern $\tau_0'.a_1(\tau_1)$ would have to be used.

Figure 7 summarizes the typing judgements for patterns. There are two kinds of judgements: $\models_P b : \tau$ (rules of the form [Qual-...Consec-T]), and $\models_P^{\diamond} b : \tau$ (remaining rules). In short, a judgement $\models_P b : \tau$ asserts that pattern b refers to an instance of type τ. The second kind of judgement $\models_P^{\diamond} b : \tau$ is necessary to deal with the case of patterns $b'.\tau$ ending in a type τ which may only apply to attributes nested further down from b'. This case is specifically dealt with by [Qual-Type-T]. For (sub-)patterns ending in attribute qualifiers, both judgements coincide ([Qual-Attr-T]).

We introduce another kind of judgement, which for a given sound pattern b (according to $\models^{\diamond}$) at a given path $\langle \tau_0 ... \tau_n, a_1 ... a_n \rangle$ determines whether b applies to that path. Figure 8 summarizes corresponding judgements of the form $\overline{\tau}, \overline{a} \Vdash_P b$ where b represents a pattern and $\langle \overline{\tau}, \overline{a} \rangle$ a fully qualified path.

In [Qual-Attr-A] we need not verify whether τ_n has an attribute a of type τ; the fact that the pattern is sound (see $\models^{\diamond}$) and that such a data structure could be constructed (which relies on type checking outlined below) ensures this. Similarly, in [Qual-Type-A] we need not verify whether $\overline{q}$ reveals an attribute a of some type τ' which is a super-type of the dynamic type τ.

The following theorem is relevant for sound t-rule selection:

Theorem 1 (Total Order on $\trianglelefteq_P$). *For any process P and fully qualified path $\langle \overline{\tau}, \overline{a} \rangle$ and any set of well-typed applicable patterns $\overline{b}$ for that path, $\trianglelefteq_P$ forms a total order on $\overline{b}$.*

By the definition of patterns, $\trianglelefteq_P$ (see Figure 11), and pattern applicability $\Vdash_P$, $\trianglelefteq_P$ on $\overline{b}$ is antisymmetric, transitive, and total.

Corollary 1 (Uniqueness). *For any fully qualified path $\langle \overline{\tau}, \overline{a} \rangle$ and any non-empty set of applicable patterns $\overline{b}$ for that path, there is exactly one pattern $b \in \overline{b}$ s.t. $\forall b' \in \overline{b} \quad b \trianglelefteq_P b'$.*

$$\frac{\tau \preceq_P \tau' \qquad \overline{\tau}, \overline{a} \Vdash_P \overline{q}}{\overline{\tau}\tau, \overline{a}a \Vdash_P \overline{q}.\tau'}\ \text{[Qual-Type-A]} \qquad \tau', \emptyset \Vdash_P \tau\ \text{[Qual-First-A]} \qquad \frac{\tau \preceq_P \tau' \qquad \overline{\tau}, \overline{a} \Vdash_P \overline{q}}{\overline{\tau}\tau, \overline{a}a \Vdash_P \overline{q}.a(\tau')}\ \text{[Qual-Attr-A]}$$

Figure 8: Pattern applicability $\overline{\tau}, \overline{a} \Vdash_P b$

4.3 Expression Typing

Next we outline type checking of expressions. Figure 9 presents the corresponding judgements $\Gamma \vdash_P e{:}\tau$, stating that in type environment Γ the expression e is of type τ. A typing environment Γ consists in pairs of the form $x{:}\tau$ where τ is the assumed type of variable x ([Env-T]).

The type of a nested object is verified by [Val-T]. [Attr-T] determines the type of an attribute of such an object, and [Call-T] straightforwardly matches the types of the actual arguments with the formals in a function call, and assigns the declared return type as type for the call expression. A publication expression is typed as $\bot$ ([Pub-T]) since the action of publishing is asynchronous and thus the publication will be reduced to an empty record $\bot[]$. Noteworthy in this rule is that a publication is only valid if the there is a valid mapping for outgoing notifications of the type of the published expression.

$$\frac{\mathit{ftype}_P(f) = \overline{\tau} \rightarrow \tau \quad \Gamma \vdash_P \overline{e} : \overline{\tau}' \quad \overline{\tau}' \preceq_P \overline{\tau}}{\Gamma \vdash_P f(\overline{e}) : \tau'}\ \text{[FCall-T]} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_P x : \tau}\ \text{[Env-T]} \qquad \frac{\mathit{attrs}_P(\tau) = \langle \overline{a}, \overline{\tau} \rangle \quad \Gamma \vdash_P \overline{e} : \overline{\tau}' \quad \overline{\tau}' \preceq_P \overline{\tau}}{\Gamma \vdash_P \tau[\overline{e}] : \tau}\ \text{[Val-T]}$$

$$\frac{\Gamma \vdash_P e : \tau \qquad \mathit{map}^!_P(\tau) = \gamma}{\Gamma \vdash_P \textbf{publish}(e) : \bot}\ \text{[Pub-T]} \qquad \frac{\Gamma \vdash_P e : \tau \qquad \mathit{attrs}_P(\tau) = \langle \overline{a}, \overline{\tau} \rangle}{\Gamma \vdash_P e.a_i : \tau_i}\ \text{[Attr-T]}$$

Figure 9: Type inference rules $\Gamma \vdash_P e{:}\tau$

4.4 Well-typed Processes and Contexts

We describe the remainder of the type system in a top-down manner. Correctness of a process P and its associated context are asserted by [Proc-OK]. The rule requires that all constituents of the process P be sound in their own respective ways, including the context c ([Ctxt-OK]) per se and subscriptions s ([Sub-OK]). In addition, the body of the process obviously needs to be well-typed according to our typing judgments as described in Section 4.3.

[Sub-OK] ensures that there is a mapping for a subscribed type and that the corresponding reaction is well-typed. [Ctxt-OK] is predicated on all elements of a context definition being well-typed. For functions, [Fun-OK] ensures that the type of the return value (implicitly formed from the function's body) respects the prognosed return type assuming that the actual arguments respect the formals. Correctness of t-rules is asserted by [Rule-OK] for transformations of incoming and outgoing notifications. The rule uses pattern typing judgments $\models_P^{\diamond} b : \tau$ described in Section 4.2 and ensures that the type of the expression qualified by the pattern (asserted via $\models_P^{\diamond}$) conforms to the formal argument type of the function used for the transformation. Note that this does not contradict the relaxed conformance put forward in Section 4.1: if the corresponding t-rule is overridden by one with a pattern which is *more specific* for that object, then our priorities imply that the pattern denotes the same type as the present one (but has an attribute instead of a type qualifier), or denotes a subtype and thus needs to conform to the present one anyway (in constrast, two functions applied by competing patterns can *return* entirely unrelated types).

4.5 Core Rules for T-Rule Resolution

Finally, [Map-OK] and [Transf] below constitute the core of the type system and t-rule resolution. Both rules rely on resolution judgements $\tau_0 ... \tau_n, a_1 ... a_n \Vdash_P^h \langle \tau, \overline{y} \rangle$ (see Figure 11), which yield two things: (1) the type τ of any expression returned by the resolved transformation at the path $\langle \tau_0 ... \tau_n, a_1 ... a_n \rangle$, and (2) a set of resolved t-rules in the form $y = \langle \tau_0 ... \tau_n, a_1 ... a_n, f \rangle$, indicating the application of f at the respective path as outlined in Section 4.1. (1) is used

$$\frac{\begin{array}{c} c\ OK\ in\ P \\ \overline{s}\ OK\ in\ P \\ \emptyset \vdash_P e : \tau \end{array}}{P(\bar{l}, c, \overline{s}, e)\ OK}\ \text{[Proc-OK]} \qquad \frac{\begin{array}{c} \overline{x} : \overline{\tau} \vdash_P e : \tau' \quad \tau' \preceq_P \tau \\ ftype_P(f) = \overline{\tau} \to \tau \end{array}}{f(\overline{x}\ \overline{\tau})\{e\}\ OK\ in\ P}\ \text{[Fun-OK]} \qquad \frac{\begin{array}{c} \overline{d}\ OK\ in\ P \\ \overline{u}\ OK\ in\ P \\ \overline{r}\ OK\ in\ P \end{array}}{(\overline{d}, \overline{u}, \overline{r})\ OK\ in\ P}\ \text{[Ctxt-OK]}$$

$$\frac{\begin{array}{c} \exists \gamma\ map^{?}_{P}(\gamma) = \tau \\ x : \tau \vdash_P e : \tau' \quad \tau \neq \bot \end{array}}{\textbf{subscribe}(\tau\ x)\{e\}\ OK\ in\ P}\ \text{[Sub-OK]} \qquad \frac{\begin{array}{c} \models^{\diamond}_{P} \tau.\overline{q} : \tau' \quad \tau' \preceq_P \tau'' \\ ftype_P(f) = \tau'' \to \tau''' \end{array}}{\tau.\overline{q} \rhd^{h} f\ OK\ in\ P}\ \text{[Rul-OK]}$$

Figure 10: Well-typed processes

by [Map-OK] to assert for a given mapping (of in- or out-bound notifications) that the type resulting from transformation corresponds to the type stated by the mapping. (2) is retained via the $tapply^{h}_{P}(\tau_0 ... \tau_n, a_1 ... a_n)$ function which maps paths to functions f ([Transf]), guiding transformation upon evaluation as explained shortly.

$$\frac{\tau, \emptyset \Vvdash^{h}_{P} \langle \tau'', ... \rangle \quad \tau'' \preceq_P \tau'}{\tau \Rrightarrow^{h} \tau'\ OK\ in\ P}\ \text{[Map-OK]} \qquad \frac{\begin{array}{c} \tau, \emptyset \Vvdash^{h}_{P} \langle ..., \overline{y} \rangle \\ \overline{y} = ... \langle \tau_0 ... \tau_n, a_1 ... a_n, f \rangle ... \end{array}}{tapply^{h}_{P}(\tau_0 ... \tau_n, a_1 ... a_n) = f}\ \text{[Transf]}$$

By the structure of [...-Transf-T] rules which use $\Vvdash$ in their premises, resolution proceeds in a recursive, top-down, manner, following the nested structure of notifications, i.e., from the top of a notification type to its leaves, as detailed hereafter. Types and any corresponding adjustments are verified at every nesting level. As outlined in Section 4.1, these rules also recursively scans subtypes.

Below we define the detailed resolution semantics captured by $\Vvdash$, summarized in Figure 11, in a stepwise manner. We first outline a helper function though. $patt_P(\overline{\tau}, \overline{a}, q)$ computes the set of patterns $b.q$ of existing t-rules ($rule^{h}_{P}(b.q) \neq \bullet$) ending in a specific qualifier q and applying to the path qualified by the first two arguments ($\overline{\tau}, \overline{a} \Vdash_P b.q$). In the case of patterns ending in an attribute qualifier the corresponding attribute and its type are also used to qualify the path ([Patters-A]), whereas due to nesting captured by patterns ending in types a corresponding trailing qualifier is not taken into account ([Patterns-T]).

1. If we have a t-rule that applies to an event type τ as a whole, we retain the corresponding function f, which is responsible for returning an expression of the expected mapped type $map^{h}_{P}(\tau)$ ([Ev-Transf-T]). The rule recursively evaluates $\Vvdash$ for any direct descendant $\tau_j \prec^{1}_{P} \tau$ of the type τ *s.t.* τ_j *maps to the same type* $map^{h}_{P}(\tau)$ *as* τ. Any t-rules resolved for such subtypes τ_j are retained ($\overline{y}_j$), and aggregated with the ones ($\overline{y}$) for τ specifically. The expected return type is the *least upper bound* (lub – $\sqcap$) of (i) all return types of such subtype τ_j transformations and (ii) the return type of f applied to the present type τ. This lub is then tested for its conformance to the expected type of the transformed notification by [Map-OK] as explained; since transformations for all subtypes, recursively, need to abide to the same mapping it is sufficient to test the lub for its conformance to the mapping in lieu of all possible resulting types.

2. If we are in a nested object (length of the pattern $n+1 \geq 2$), we take the prioritary t-rule for the current path, if any ([Attr-Transf-T]). That is, several competing t-rules may exist for the current path. The corresponding patterns $\overline{b}'$ are obtained via $patt_P(\overline{\tau}, \overline{a}, q)$. These end either (i) exactly in the last attribute of the path (see $\overline{b}$), or (ii) in the type of that attribute with a pattern corresponding to a prefix of any length i of the path ($\overline{b}' \backslash \overline{b}$). We choose the rule $rule^{h}_{P}(b)$ corresponding to the pattern b which is prioritary over any (other) pattern b' ($b \trianglelefteq b'$) in the set $\overline{b}'$. By the absence of multiple t-rules with identical patterns and the properties of $\trianglelefteq$ (see Corollary 1), there is exactly one such pattern in a non-empty set of applicable patterns.

   Note that just like in the case of 1., we (recursively) perform the same resolution for all direct subtypes τ'_j of τ_n, and return the union of all correspondingly resolved t-rules and the lub of the types resulting from the respective transformations.

3. If no t-rule has a pattern matching the path and we are transforming a leaf (characterized by the absence of attributes $attrs_P(\tau_n) = \langle \emptyset, \emptyset \rangle$) we leave the attribute unchanged ([Val-Transf-T]). Treatment of subtypes of τ_n occurs as before.

4. If no t-rule applies but we are transforming a nested object ([Nested-Transf-T]) we proceed recursively by resolving t-rules ($\overline{y}_k$) for any attribute a'_k nested at the current path, *as well as* recursing into any subtype τ'''''_s of type τ_n ($\overline{y}_s$).

5. TRANSFORMATION APPLICATION

Now that we have described how t-rules are pre-resolved ($\overline{\tau}, \overline{a} \Vvdash^{h}_{P} \langle \tau, \overline{y} \rangle$) and retained ($tapply^{h}_{P}(\overline{\tau}, \overline{a})$), we proceed to describing their application at runtime by characterizing program evaluation as a contextual operational semantics [22].

5.1 Contextual Operational Semantics

E represent evaluation contexts with the following grammar:

$$E ::= [] \mid \overline{v}\, E\, \overline{e} \mid \tau[E] \mid f(E) \mid \textbf{publish}(E)$$

In Figure 12, which presents the evaluation rules, $\longrightarrow^{P}$ is the local evaluation relation defined on expressions e in a specific context P ($\longrightarrow^{P}$ defines redexes), while $\longrightarrow$ is defined on programs Π. As usual, a congruence links the two relations. In this case ([Congruence]), the congruence applies to both a process' body as well as to its outgoing notifications.

[F-Call] reduces function calls to substitution $^{\overline{v}}/_{\overline{x}}$ of actual arguments $\overline{v}$ for formal ones $\overline{x}$ in the function's body e ($\{^{\overline{v}}/_{\overline{x}}\}e$). Attribute access is described by [A-Acc]. [Ev-Publ] handles the publication of an event e by reducing the publication expression to $\bot[]$ (to model asynchrony of interaction), and placing the published expression — resulting

$$\begin{gathered} patt^h_P(\overline{\tau}, \overline{a}, a(\tau)) = \\ \{b.a(\tau') \mid \tau \preceq_P \tau' \wedge \overline{\tau}\tau', \overline{a}a \Vdash_P b.a(\tau') \wedge rule^h_P(b.a(\tau')) \neq \bullet\} \end{gathered} \quad \text{[PATTS-A]}$$

$$patt^h_P(\overline{\tau}, \overline{a}, \tau) = \{b.\tau' \mid \tau \preceq_P \tau' \wedge \overline{\tau}, \overline{a} \Vdash_P b \wedge rule^h_P(b.\tau') \neq \bullet\} \quad \text{[PATTS-T]}$$

$$\frac{\begin{gathered}\tau_1 ... \tau_w = \{\tau' \mid \tau' \prec^1_P \tau \wedge map^h_P(\tau') = map^h_P(\tau)\} \quad rule^h_P(\tau) = f \\ \forall j \in [1..w] \tau_j, \emptyset \Vvdash^h_P \langle \tau'_j, \overline{y}_j \rangle \quad ftype_P(f) = ... \to \tau' \end{gathered}}{\tau, \emptyset \Vvdash^h_P \langle \sqcap \overline{\tau}' \; \tau', \langle \tau, \emptyset, f \rangle \cup \bigcup_{j \in [1..w]} \overline{y}_j \rangle} \quad \text{[EV-TRANSF-T]}$$

$$\frac{\begin{gathered}\overline{b} = patt^h_P(\tau_0 ... \tau_{n-1}, a_1 ... a_{n-1}, a_n(\tau_n)) \quad b \in \overline{b}' \mid \forall b' \in \overline{b}' \; b \trianglelefteq_P b' \\ \overline{b}' = \overline{b} \cup \bigcup_{i \in [0..n]} patt^h_P(\tau_0 ... \tau_{i-1}, a_1 ... a_{i-1}, \tau_n) \quad \overline{b}' \neq \emptyset \\ \tau'_1 ... \tau'_w = \{\tau \mid \tau \prec^1_P \tau_n\} \quad ftype_P(f) = ... \to \tau'' \quad rule^h_P(b) = f \\ n \geq 1 \quad \forall j \in [1..w] \tau_0 ... \tau_{n-1} \tau'_j, a_1 ... a_n \Vvdash^h_P \langle \tau''_j, \overline{y}_j \rangle \end{gathered}}{\tau_0 ... \tau_n, a_1 ... a_n \Vvdash^h_P \langle \sqcap \overline{\tau}'' \; \tau'', \langle \tau_0 ... \tau_n, a_1 ... a_n, f \rangle \cup \bigcup_{j \in [1..w]} \overline{y}_j \rangle} \quad \text{[ATTR-TRANSF-T]}$$

$$\frac{\begin{gathered}patt^h_P(\tau_0 ... \tau_{n-1}, a_1 ... a_{n-1}, a_n(\tau_n)) = \emptyset \quad \tau'_1 ... \tau'_w = \{\tau \mid \tau \prec^1_P \tau_n\} \\ \bigcup_{i \in [0..n]} patt^h_P(\tau_0 ... \tau_{i-1}, a_1 ... a_{i-1}, \tau_n) = \emptyset \quad attrs_P(\tau_n) = \langle \emptyset, \emptyset \rangle \\ n \geq 1 \quad \forall j \in [1..w] \tau_0 ... \tau_{n-1} \tau'_j, a_1 ... a_n \Vvdash^h_P \langle \tau''_j, \overline{y}^j \rangle \end{gathered}}{\tau_0 ... \tau_n, a_1 ... a_n \Vvdash^h_P \langle \sqcap \overline{\tau}'' \; \tau_n, \bigcup_{j \in [1..w]} \overline{y}_j \rangle} \quad \text{[VAL-TRANSF-T]}$$

$$\frac{\begin{gathered}patt^h_P(\tau_0 ... \tau_{n-1}, a_1 ... a_{n-1}, a_n(\tau_n)) = \emptyset \quad attrs_P(\tau''_n) = \langle \overline{a}', \overline{\tau}''' \rangle \\ \bigcup_{i \in [0..n]} patt^h_P(\tau_0 ... \tau_{i-1}, a_1 ... a_{i-1}, \tau_n) = \emptyset \\ \exists \tau''_0 ... \tau''_n \mid \forall j \in [1..n] \; attrs_P(\tau''_{j-1}) = \langle ... a_j ..., ... \tau''_j ... \rangle \wedge map^h_P(\tau_0) = \tau''_0 \\ n \geq 1 \quad attrs_P(\tau_n) = \langle a'_1 ... a'_z, \overline{\tau}' \rangle \quad \tau'''''_1 ... \tau'''''_w = \{\tau \mid \tau \prec^1_P \tau_n\} \\ \forall k \in [1..z] \; \tau_0 ... \tau_n \tau'_k, a_1 ... a_n a'_k \Vvdash^h_P \langle \tau''''_k, \overline{y}_k \rangle \wedge \tau''''_k \preceq_P \tau'''_k \\ \forall s \in [1..o] \tau_0 ... \tau_{n-1} \tau'''''_s, a_1 ... a_n \Vvdash^h_P \langle \tau''''''_s, \overline{y}'_s \rangle \wedge \tau''''''_s \preceq_P \tau''_n \end{gathered}}{\tau_0 ... \tau_n, a_1 ... a_n \Vvdash^h_P \langle \tau''_n, \bigcup^z_{k=1} \; \overline{y}_k \cup \bigcup^o_{s=1} \; \overline{y}'_s \rangle} \quad \text{[NESTED-TRANSF-T]}$$

Figure 11: Determining t-rules

from applying the corresponding transformation (identified earlier by $\Vvdash^!_P$) to it via $\mathcal{T}^!_P[\![e]\!]_{\langle ... \rangle}$ — in the queue of outgoing expressions. After transformation of the notification completes, [EV-MCAST] places it into the "environment" (the network). Inversely, [EV-DLVR] takes a notification in the environment and, for all processes $\overline{p}'$ interested in the respective event type, adds a corresponding reaction to the process bodies ([EV-DLVR]). Such a reaction is obtained by looking up the appropriate reaction expression of a considered process P for the event type, and substituting the notification m with transformation $\mathcal{T}^?_P[\![m]\!]_{\langle \rangle}$ applied to it for the formal argument of the reaction expression.

$$P\langle ..., m \, \overline{e} \rangle \parallel \overline{m} \; \parallel ... \longrightarrow P\langle ..., \overline{e} \rangle \parallel \overline{m} \, m \; \parallel \; ... \quad \text{[EV-MCAST]}$$

$$P\langle E[\textbf{publish}(\tau[\overline{v}])], \overline{e} \rangle \parallel ... \longrightarrow P\langle E[\bot[\,]], \overline{e} \, \mathcal{T}^!_P[\![\tau[\overline{v}]]\!]_{\langle \tau, \emptyset \rangle} \rangle \parallel ... \quad \text{[EV-PUBL]}$$

$$\frac{\begin{gathered}\overline{p}' = \{P\langle ..., ... \rangle \in \overline{p} \mid \exists \gamma' \; \gamma \preceq_P \gamma' \wedge \gamma' \in subs_P\} \\ \forall P\langle ..., ... \rangle \in \overline{p}' \; \tau_p = map^?_P(\gamma) \wedge react_P(\tau_P) = \langle x_P, e_P \rangle \\ \overline{p}'' = \{P\langle \overline{e} \, \{^{\mathcal{T}^?_P[\![\gamma[\overline{v}]]\!]_{\langle \gamma, \emptyset \rangle}}/_{x_P}\} e_P, ... \rangle \mid P\langle \overline{e}, ... \rangle \in \overline{p}'\} \end{gathered}}{\overline{g} \parallel \overline{p} \parallel \gamma[\overline{v}] \, \overline{m} \longrightarrow \overline{g} \parallel \overline{p}'' \parallel \overline{p} \backslash \overline{p}' \parallel \overline{m}} \quad \text{[EV-DLVR]}$$

$$\frac{fbody_P(f) = \langle \overline{x}, e \rangle}{f(\overline{v}) \longrightarrow^P \{^{\overline{v}}/_{\overline{x}}\} e} \; \text{[F-CALL]} \qquad \tau[e_1 ... e_n].a_i \longrightarrow^P e_i \; \text{[A-ACC]}$$

$$\frac{e \longrightarrow^P e'}{P\langle ... E[e] ... \rangle \parallel ... \longrightarrow P\langle ... E[e'] ... \rangle \parallel ...} \quad \text{[CONGR]}$$

Figure 12: Core evaluation semantics

5.2 T-Rule Application

Transformation $\mathcal{T}^h_P[\![m]\!]_{\langle \overline{\tau}, \overline{a} \rangle}$ occurs following the nested structure of notifications m, i.e., from the top of a notification to its leaves, analogously to t-rule resolution. At path $\langle \overline{\tau}, \overline{a} \rangle$, we apply the t-rules retained in $tapply^h_P(\overline{\tau}, \overline{a})$, by following the rules presented in Figure 13. [EV&ATTR-TRANSF] describes the transformation at a given path $\langle \overline{\tau}, \overline{a} \rangle$ for which a transformation function has been determined. [VAL-TRANSF] deals with the case where there is no such function and the currently considered value is a primitive one. (In practice we can recognize such branches earlier and avoid even getting here.) Finally, [NESTED-TRANSF] describes the case of nested transformation, which leads to exploring every derived path recursively in the order of declaration of the attributes. To respect the high-level type mapping, a type change (to τ''_n) might be performed when returning back up based on the mapping of the notification type τ_0.

$$\frac{tapply^h_P(\overline{\tau}, \overline{a}) = \bullet}{\mathcal{T}^h_P[\![\tau[\,]]\!]_{\langle \overline{\tau}, \overline{a} \rangle} = \tau[\,]} \; \text{[VAL-TRANSF]} \qquad \frac{tapply^h_P(\overline{\tau}, \overline{a}) = f}{\mathcal{T}^h_P[\![v]\!]_{\langle \overline{\tau}, \overline{a} \rangle} = f(v)} \; \text{[EV\&ATTR-TRANSF]}$$

$$\frac{\begin{gathered} w > 1 \quad map^h_P(\tau_0) = \tau''_0 \\ attrs_P(\tau_n) = \{a'_1 ... a'_w, \tau'_1 ... \tau'_w\} \quad tapply^h_P(\tau_0 ... \tau_n, \overline{a}) = \bullet \\ \tau''_1 ... \tau''_n \mid \forall j \in [0..n-1] \; attrs_P(\tau''_j) = (... a_{j+1} ..., ... \tau''_{j+1} ...) \end{gathered}}{\begin{gathered}\mathcal{T}^h_P[\![\tau_n[v_1 ... v_w]]\!]_{\langle \tau_0 ... \tau_n, \overline{a} \rangle} = \\ \tau''_n[\mathcal{T}^h_P[\![v_1]\!]_{\langle \tau_0 ... \tau_n \tau'_1, \overline{a} a'_1 \rangle}, ..., \mathcal{T}^h_P[\![v_w]\!]_{\langle \tau_0 ... \tau_n \tau'_w, \overline{a} a'_w \rangle}]\end{gathered}} \quad \text{[NESTED-TRANSF]}$$

Figure 13: Transformation application

5.3 Type Safety

Now we can proceed to stating type safety for our language based on the type system and evaluation semantics presented.

THEOREM 2 (PROGRESS). *Suppose e is a closed well-formed normal form. Then e is a value.*

THEOREM 3 (PRESERVATION). *If P is a well-typed process, $\Gamma \vdash_p e : \tau$, and $\Pi || P\langle ... E[e] ... \rangle \longrightarrow \Pi || P\langle ... E[e'] ... \rangle$ then $\Gamma \vdash_p e' : \tau'$ such that $\tau' \preceq_P \tau$.*

Theorem 3 follows the common form of preservation theorems, which may seem surprising at first glance in the presence of transformations including type adaptations. However, transformations on any given process P are applied to both incoming and outgoing notifications through function $\mathcal{T}_P[\![...]\!]_{\langle ... \rangle}$ in an *atomic* manner *regarding their types*, and only when/before they are added to the sequence of expressions representing P's body or outgoing publications respectively. Thus any expression in any of the two "threads" of a process maintains its type.

Note though that while types are transformed atomically (when returning recursively from the nested transformation

– see [Nested-Transf]) the resulting object may still contain expressions that are not fully evaluated, as values of attributes are replaced by calls to the respective transformation functions (e.g., $f(e)$).

Theorem 4 (Global Progress). *Suppose two well-typed processes P and P' s.t. $map^{!}_{P}(\tau) = \gamma$, $map^{?}_{P'}(\gamma) = \tau'$, $react_P(\tau') = \langle x, e\rangle$, and $\Pi = P\langle E[\textbf{publish}(\tau[...])], ...\rangle \parallel P'\langle ...\rangle \mid \mid ...$, then $\Pi \longrightarrow^{*} P'\langle ...\{^{\tau''[...]}/_{x}\}e, ...\rangle \parallel ...$ s.t. $\tau'' \preceq_P \tau'$ with $\longrightarrow^{*}$ the transitive closure of $\longrightarrow$.*

As we keep applying the rules of Figure 12, an expression published by P and "out-mapped" to a type which is "in-mapped" by P' will eventually be delivered in an appropriately transformed manner to P'. The choice to model in-transit multicast messages as a sequence as in unicast scenarios [12] despite the increased implementation hardness here is only in support of this theorem; otherwise we could modify [Ev-Dlvr] to deliver messsages in arbitrary order without affecting our semantics.

6. RECURSIVE TYPES

This section describes simple extensions to the previously presented model in order to deal with (mutually) recursive types. Figure 14 summarizes all necessary extensions. Definitions or rules named X_{rec} *replace* previous definitions or rules X.

6.1 Design and Syntax

We adopt an approach where transformations are *insensitive* to the depth of recursion. For example, we prohibit patterns like $\tau_0.\tau_1.\tau_1$ with τ_1 a recursive type containing one or more attributes of τ_1. While such t-rules could provide additional flexibility they jeopardize simplicity: typically programming languages do not assign special semantics to the actual level of recursion.

The only syntactic extension necessary for dealing with recursive types aims at being able to limit recursion in expressions. To that end, we simply introduce a **null** value:

$$value_{rec} \quad v \quad ::= \quad m \quad | \quad \textbf{null}$$

Such a value can be used as a value of any type ([Null-T] in Figure 14). This seemingly small extension also explains why we introduce recursive types separately: our previous theorems for progress will trivially be violated.

6.2 T-rule Resolution and Application

When identifying applicable transformations for a path ending in a recursion we can make the same simplifying observation as earlier: if there was a t-rule applying to the same path modulo the recursive part then that t-rule would have been identified already for that prefix and the corresponding transformation would have been applied, preventing us from reaching this point. As a consequence, when discovering a recursion in a fully qualified path, we can simply assume that the same t-rules (if any) will apply to the individual attributes recursively and we can halt the resolution at this point. This corresponds to the *unfolding* of recursion

$$\frac{\exists i \; recur(\tau_0 ... \tau_n, a_1 ... a_n, i, n)}{\tau_0 ... \tau_n, a_1 ... a_n \Vdash^{h}_{P} \langle \top, \emptyset \rangle} \; \text{[Rec-Transf-T]} \qquad \frac{i < j \quad \tau_i \preceq_P \tau_j}{recur(\tau_0 ... \tau_n, a_1 ... a_n, i, j)} \; \text{[Cycle]}$$

$$\Gamma \vdash_P \textbf{null} : \tau \; \text{[Null-T]}$$

$$\frac{\exists i recur(\tau_0 ... \tau_n, a_1 ... a_n, i, n)}{\begin{array}{c} tapply^{h}_{P}(\tau_0 ... \tau_n, a_1 ... a_n) = \\ tapply^{h}_{P}(\tau_0 \tau_{i-1} \tau_n, a_1 ... a_{i-1} a_n) \end{array}} \; \text{[Rec-Transf]} \qquad \frac{tapply^{h}_{P}(\overline{\tau}, \overline{a}) = \bullet}{\mathcal{T}^{h}_{P}[\![\textbf{null}]\!]_{\langle \overline{\tau}, \overline{a} \rangle} = \textbf{null}} \; \text{[Null-Transf]}$$

$$\frac{\dfrac{A}{C} \; [R\text{-Transf-T}] \qquad R \in \{\text{Attr}, \text{Nested}, \text{Val}\}}{\dfrac{A \quad \nexists i \; recur(\overline{\tau}, \overline{a}, i, n)}{C} \; [R\text{-Transf-T}_{rec}]} \; \text{[Rec-Meta-Transf-T]}$$

Figure 14: Extended syntax and rules for dealing with recursive types. Definitions of the form Def_{rec} replace the respective definitions Def. Rule [Rec-Meta-Transf-T] shows how to simply extend the respective rules from Figure 11

with *iso-recursive* types. [Rec-Transf-T] captures this. The rule makes use of the definition of a cycle of types ($recur()$, [Cycle]) of which each has an attribute of a super-type of the next type in the cycle, which represents the general case leading to recursion here.

Note that we introduce a "top" type $\top$ which is a subtype of any type, thus turning any type hierarchy in our language into a lattice. $\top$ is not actually used to ever type anything, but only to represent the absence of a t-rule and thus a target type for the transformation. Whenever taking the lub of $\top$ any other type τ ($\sqcap \, \tau \, \top$), we trivially obtain τ. If no other t-rule is identified then $\top$ will allow us to correctly pass the conformance check with any mapped type ([Map-OK]). At runtime, this case means that no transformation is taking place so no type conversion to $\top$ will be attempted.

Several rules from Figure 11 have to be augmented to avoid conflicts with [Rec-Transf-T]. The extension is however simple and always the same, consisting simply in verifying the absence of a recursion in the current path. To save space, rather than repeating the respective rules with the small change, we represent the changes through a "meta-rule" ([Rec-Meta-Transf-T]) that describes substitute rules [...-Transf-T$_{rec}$] for the respective [...-Transf-T] rules: the same premise $\nexists i \; recur(\overline{\tau}, \overline{a}, i, n)$ for testing absence of cycles is added to the respective existing preconditions A while leaving the corresponding conclusion C unchanged.

Based on the above observation, when applying t-rules at a given path exhibiting a recursion ([Rec-Transf]), we only need to consider t-rules for the same path *without* the recursive part (*folding*). Finally, [Null-Transf] is necessary to deal with **null** values at runtime to avoid further nested transformation attempts.

6.3 Type Safety

The following progress theorem is adapted from Theorem 2 to account for **null** values introduced to support recursive types.

Theorem 5 (Recursion Progress). *Suppose e is a closed well-formed normal form. Then e is a value or $\exists E \mid$*

$e = E[\textbf{null}.a]$.

This theorem does not distinguish between (a) any **null**.a expressions being generated — by mistake — by the transformation evaluation rules or (b) those generated by programs (e.g., by absence of non-**null** checks in process and reaction bodies, or from transformation functions f returning such values). The only new evaluation rules introduced to support recursive types are [REC-TRANSF] and [NULL-TRANSF]. Since it is easy to see that these rules do not introduce expressions of the form **null**.v themselves, and inherited rules are used in fewer cases ([REC-META-TRANSF-T]), the only cause for such expressions are (b).

7. IMPLEMENTATION AND PERFORMANCE

In this section we show that the native implementation of our approach is (a) much faster than an analogous library implementation based on reflection and (b) as effective as a manually coded transformations in application components. We also show that (c) the application of transformations closer to producers, enabled by our eager resolution, further improves performance. We use a micro-benchmark, the SPECjms2007 [3] benchmark, and a benchmark with a typical workload from our motivating applications.

7.1 ACTrESS

ACTrESS ("Automatic Context Transformation for Event-based Software Systems") implements our approach for the Java programming language. ACTrESS is built on top of ActiveMQ [21], a fast, reliable JMS [4] broker. Our approach is implemented as a *plugin* for easy adaptation to other systems or languages. It intercepts event notifications passing through the broker and transforms them according to a t-rule set. Functions f used by transformations are methods invoked on notifications or their attributes, or **static** methods. Our prototype generates a class containing transformation code after analyzing t-rules and notification types.

7.2 Evaluation Setup

We compare our approach in the following against a library-based approach in which each incoming notification is analyzed with Java reflection, via which transformations are applied accordingly. This is how a library implementation would work, as it has to process notifications of (yet) unknown types. On the upside, this allows new types to be straightforwardly added at runtime. However, our experience shows that new or changed notification types are rare compared to the number of notifications that are processed. In addition, upon encountering a new notification type in a given process, the corresponding class has to be downloaded regardless before a corresponding instance can be used, and this gives time to recompile and dynamically load the plugin in our case.

To better analyze sources of overhead we ran experiments in a distributed setup as well as in a local one. For the distributed setup, we ran the ACTrESS broker (cf. *communication substrate*, Figure 1) on a server with two Intel Xeon Quad-Core with 2.33GHz each and 16GB RAM. Workload producers ($prod_i$) and several data collectors ($cons_i$) were run on a server with four Intel Xeon Dual-Core with 3.4GHz each and 16GB RAM. Using the more powerful machine for workload production ensures that performance results are not influenced by a weak client. In the local setting, broker and clients ran on the same machine. We use this setting to see beyond network contention and latency.

We compared four setups: in *none*, brokers just access notification content (as in current content-based notification systems) but do not perform any transformations, assuming all parties agree upfront on types; *compile* transforms notifications according to our approach, while *reflect* uses reflection for transformation resolution and application. Finally, in the *base*line case brokers simply forward notifications without accessing their content.

Table 1: Maximum throughput (in 1000 notifications/s)

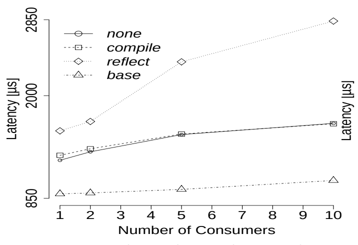
	Generated	SPECjms2007	Logistics
base	22.5 (100%)	16.8 (100%)	63.7 (100%)
none	20.8 (92%)	14.1 (84%)	55.4 (87%)
compile	20.7 (92%)	13.4 (80%)	53.6 (84%)
reflect	11.2 (50%)	5.4 (32%)	23.1 (36%)

7.3 Micro-Benchmark

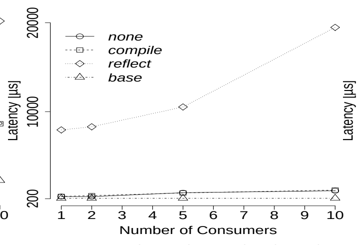
The micro-benchmark uses a generated workload to see the influence of various parameters on the performance and push the system to its limits. We let our experience from the projects mentioned in Section 1 guide the benchmark design to get a well-grounded workload: we assume up to ten consumers per producer that need to have events transformed to correctly interpret them. This is a realistic value for a logistics provider operating world-wide. If there are more consumers for a given producer, intermediate nodes (e.g., brokers) are typically used as relays.

Native support vs. reflection.. The maximum throughput for *none* and *compile* is nearly the same, with differences within measurement error. The maximum throughput for *reflect* is about half as much across all configurations. *base* achieves about 8% more throughput than *none* or *compile* (see Table 1 for details). Figure 15a shows the latencies of the different scenarios with varying numbers of consumers. Again, there is no observable difference between *none* and *compile*. *reflect* however introduces additional latency which grows in relation to the other two approaches, reducing its scalability with respect to our approach. *base* has only half the latency of *none* and *compile*, as it need not access notification content. All three approaches scale similarly.

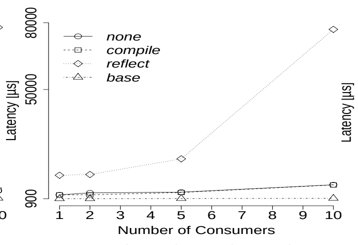
To better dissect the results, we ran the same experiment in a local setting (see Figure 15b) with notification system and producers/consumers on the same machine. As expected, the maximum throughput is generally a little lower than in a distributed environment because now one machine has to do the whole work, leading to local contention. The relative throughput performance between the scenarios is the same as above. The latency analysis however shows that the difference between *reflect* and the other two scenarios is much higher now, with *reflect* being ≈10 times slower. Major parts of this overhead are thus masked by network latency but as illustrated in the distributed setup the difference is still clearly measurable. This analysis also shows that our approach is as effective as manual transformations in application components or an a priori agreement on types, since



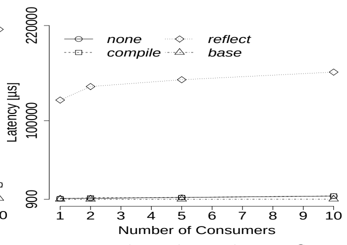
(a) Micro-benchmark: distributed



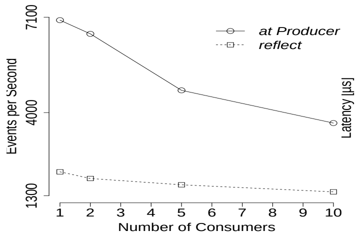
(b) Micro-benchmark: local



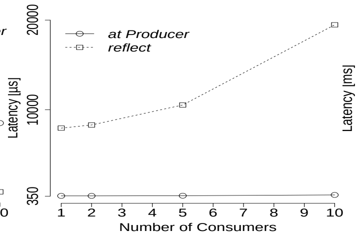
(c) Micro-benchmark: deep structure



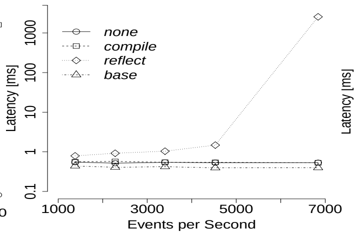
(d) Micro-benchmark: flat structure



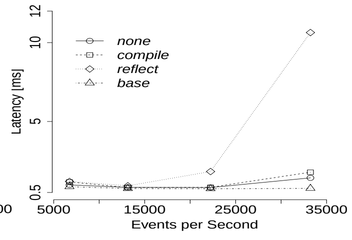
(e) Micro-b.: close (throughput)



(f) Micro-b.: close (latency)



(g) SPECjms2007-based benchmark



(h) Traffic information benchmark

Figure 15: Performance results. Confidence intervals are omitted for readability. 90% percentiles were all within 5% of the values reported

we do not add any measurable overhead with respect to no transformations taking place (*none*).

Complexity of events.. We also evaluated the influence of the complexity of event notifications. To that end we added ten attributes to the five attributes which the events already had. We added these attributes both in a "flat" manner and by increasing nesting levels. Figures 15c and 15d show the latency under the respective modifications (throughput is only marginally affected). More attributes generally increase the latency, accounting for the greater (de)serialization effort (*base* is much less effected). However, a flat event hierarchy affects *reflect* much more than the deep hierarchy (note the different scale on the ordinate). In fact, *none* and *compile* perform better with a flat hierarchy than with a deep one, while *reflect* performs better in a deep hierarchy compared to a flat one in the same approach (the steep increase at the right side of the graph does not continue for a higher consumer count). Our experience shows that events do not exhibit deep nesting levels, accentuating the advantages of our approach.

Transforming closer to the producer.. Our approach of explicitly denoting transformations allows transformations to be pushed closer to producer, which is desirable in distributed publish/subscribe systems. We thus compared the transforming of notifications close to the producer with the same transformations happening at each consumer $cons_i$. As the results show, being able to push the transformation close to the producer leads to huge gains in throughput (Figure 15e) and latency (Figure 15f).

7.4 SPECjms2007 Workload

The SPECjms2007 [3] standard benchmark specifies a workload where distribution centers, headquarters, and suppliers form a complex supply chain and interact via various inter-company notifications. A distribution center starts an interaction by sending a notification that asks for offers for a specific product; this notification triggers a number of additional notifications sent between different participants. We designed a workload following SPECjms2007. Notification types are taken directly from the specification. Transformations include addresses, distances, and currency.

Figure 15g shows the latency for different publishing rates. Please note that the ordinate has a logarithmic scale. As the figure shows, *reflect* adds significant latency overhead compared to our approach. This is an indicator for the increased computational effort of the reflection-based approach. ActiveMQ cannot cope with higher notification rates due to this additional effort; notifications are stalled as they cannot be processed right away, accounting for the steep latency increase. The start of the steep increase also indicates the throughput limit. A higher notification production rate will just cause more notifications to become stalled, resulting in exponential increase of latencies.

Scenarios *none* and *compile* achieve more than double the throughput, while *base* achieves even slightly more notifications per second (see Table 1 for details).

7.5 Traffic Information

The third benchmark is a traffic information scenario with many participants and lightweight notifications [17]. The key idea in this scenario is to monitor traffic and pass that information to interested consumers: emergency cars, police, and (for a fee) taxis or logistics providers. The logistics provider from our example thus needs to understand traffic information formats for all serviced cities.

Traffic sensors report traffic densities to a traffic information center. This center aggregates the information and sends update events to information boards distributed throughout the city. The aggregated information is also passed to cars which need up-to-date traffic information. Transformations include string conversions and translation of latitude/longitude coordinates and addresses.

The results of running this workload at different notification rates (see Figure 15h) are consistent with the previous results: our approach does not add any measurable overhead compared to the approach that does not perform any transformations. *reflect* however adds overhead which further increases with the notification rate. This results in a reduced maximum throughput.

In summary, our approach achieves the same flexibility as a reflection-based approach, but with much better performance and with better guarantees. In all investigated scenarios, analyzing the t-rule set and generating the transformations class took less than 100ms. The performance decrease of our approach with respect to the baseline can be explained almost completely by the overhead of accessing event notification content, which a more expressive (i.e., content-based) broker would have to do anyway. The difference decreases with more complex workloads, while the difference to a reflection-based approach increases.

8. RELATED WORK

Cluet et al. [9] address the issue of integrating heterogeneous data sources by proposing a rule language for conversion between various data representations. The system is designed for request/reply communication while we focus on data distributed via publish/subscribe, i.e., following a *multi*cast model. In publish/subscribe systems subscribers may not know the origin of data, thus one cannot simply compare two communication endpoints. Cilia et al. [8] propose a solution to deal with heterogeneous data sources in publish/subscribe systems. The model does not guarantee soundness.

Foster et al. [11] present a bi-directional tree transformation approach. Their transformations allow to mediate between different views of same data where updates are applied backwards to the original data. In contrast, our system purposely supports one-directional transformations. Every subscriber gets its own copy of a notification and transformed notifications are not meant to be shared unlike documents. Thus there is no requirement to propagate changes backwards to source objects.

HydroJ [13] extends Java with relaxed conformance on nested semi-structured events exchanged between processes. HydroJ focuses on unicast communication and extends the host language syntax and type system. In contrast, our approach can be employed without modifying the syntax of a host language. Due to the semantics of references and 1-1 messages in HydroJ there's a clear binding between caller and callee sites, enabling simple verification of conformance. An architecture for structural subtyping in distributed *multicast*-based systems is described in [16]. The approach refrains specifically from extending any programming language and to that end promotes the use of hash maps to convey events in the form of $\langle key, value\rangle$ pairs. This simplifies the design of the multicast infrastructure but puts all the burden on application developers as these need to manually inspect and marshal/unmarshal events at generation and reception. As with other structural typing models, semantic transformations (e.g., enrichment) are not supported.

In Hashtypes [14] type representations are hashed, including "contents" of instances, function signatures in modules, etc. These hashes are propagated at runtime together with corresponding objects. Flexibility is increased by allowing programmers to indicate which constituents are relevant for comparison (e.g., equality on module names). Hashtypes are refined in [10] with a notion of subtyping to accommodate also unidirectional transformations. The approach supports partially abstract types in the form of bounded existentials. Subtyping in this model hinges on a notion of subhashes, which are computed and disseminated at runtime. Hashtypes do not support value transformations (e.g., unit conversions) or enrichment.

Distributed extensible type encoding [6] is orthogonal to our work and can be used to extend the global type set $\overline{g}$ in practice.

Monsanto et al. [15] present a language for programming routing infrastructures supporting the paradigm of *software-defined networking*. The language is used to express packet forwarding policies, and a compiler generates code for efficient execution on switches. This approach is similar in spirit to ours yet targets a different problem domain. Packet forwarding rules are typically not able to capture transformations; inversely, our language does not aim at expressing algorithms for efficiently matching notifications to a set of different subscriptions. This is the responsibility of the application-level routing protocols in ACTrESS which are fixed.

Our process model bears superficial similarities with Actors [5]. It serves mostly as a vehicle for the introduced transformation semantics that constitute our main contribution. Similarly, we have refrained from expressing our transformation semantics through the notion of *cells* of the m-Calculus [20]. While its *membranes* could be used to capture transformations, our model is simpler yet sufficient and matches much closer the targeted object languages.

9. CONCLUSIONS

We have introduced a foundational model for networked distributed applications in which types are federated. Our model is generic in that it supports the whole spectrum of transformations including enrichment of events and allows other type conformance models to be implemented atop; it is flexible by supporting fine-grained expression of transformations as opposed to monolithic ones. Last but not least, our approach is safe, by promoting clear semantics for transformations, whose application is validated, and enforced.

We are currently investigating extensions to our model including multiple subtyping, or facilitating nested transformations by allowing the transformation process to be explicitly (re-)invoked from within transformation functions. Our implementation includes an identity function (cf. Section 2.3) which is polymorphic in that its return type is the same as that of its argument.

10. REFERENCES

[1] Amazon Simple Notification Service. `http://aws.amazon.com/sns/`.

[2] Project Kafka.
http://engineering.linkedin.com/tags/kafka.

[3] SPECjms 2007. http://www.spec.org/jms2007/.

[4] Java Message Service - Specification, version 1.1, 2008. http://www.oracle.com/technetwork/java/jms/index.html.

[5] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[6] H. Alavi, S. Gilbert, and R. Guerraoui. Extensible Encoding of Type Hierarchies. In *POPL 2008*.

[7] A. Carzaniga and A. Wolf. Forwarding in a Content-based Network. In *SIGCOMM 2003*.

[8] M. Cilia, M. Antollini, C. Bornhövd, and A. Buchmann. Dealing with Heterogeneous Data in Pub/Sub Systems: The Concept-Based Approach. In *DEBS 2004*.

[9] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your Mediators Need Data Conversion! In *SIGMOD 1998*.

[10] P.-M. Deniélou and J. J. Leifer. Abstraction Preservation and Subtyping in Distributed Languages. In *ICFP 2006*.

[11] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for Bi-directional Tree Transformations: a Linguistic Approach to the View Update Problem. In *POPL 2005*.

[12] K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL 2008*.

[13] K. Lee, A. LaMarca, and C. Chambers. HydroJ: Object-oriented Pattern Matching for Evolvable Distributed Systems. In *OOPSLA 2003*.

[14] J. J. Leifer, G. Peskine, P. Sewell, and K. Wansbrough. Global Abstraction-safe Marshalling with Hash Types. In *ICFP 2003*.

[15] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *POPL 2012*.

[16] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *SOSP 1993*.

[17] S. Schneider. DDS and the Future of Complex, Distributed Data-Centric Embedded Systems, 2006. http://www.eetimes.com/design/embedded/4025967/DDS-and-the-future-of-complex-distributed-data-centric-embedded-systems.

[18] M. Sadoghi and H.-A. Jacobsen. BE-Tree: an Index Structure to Efficiently Match Boolean Expressions over High-Dimensional Discrete Space. In *SIGMOD 2011*.

[19] D. Sangiorgi and D. Walker. *Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[20] A. Schmitt and J.-B. Stefani. The M-Calculus: a Higher-order Distributed Process Calculus. In *POPL 2003*.

[21] B. Snyder, D. Bosanac, and R. Davies. *ActiveMQ in Action*. Manning Publications Co., 2011.

[22] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.