Implementing Federated Object Systems*

Tobias Freudenreich[2], Patrick Eugster[1], Sebastian Frischbier[2], Stefan Appel[2], and Alejandro Buchmann[2]

[1] Department of Computer Science, Purdue University, USA
p@cs.purdue.edu
[2] Databases and Distributed Systems Group (DVS), TU Darmstadt, Germany
{*lastname*}@dvs.tu-darmstadt.de

Abstract. With the increase of automatically sensed and generated data in distributed software systems, the publish/subscribe paradigm gains importance. Automatically generated notifications are pushed from their publishers to interested subscribers. Interoperability is a core issue in such federated networked distributed applications. However, the problems of heterogeneity must be reconsidered in the light of tougher conditions than previously: low latency delivery in addition to expressiveness and extensibility.
To aid in engineering federated distributed systems, this paper proposes a framework for object transformations. Components can operate in individual, semantic contexts, which include local type declarations, fine-grained transformation rules (t-rules), and type mappings that express the programmer's intent at a high level. Our generic approach supports transformations at any granularity using clear priorities to select among complementary t-rules.
We present empirical evidence of the efficiency of our approach and of the benefits to the programmer in terms of code quality.

Keywords: Heterogeneity, Publish/Subscribe, Semantic Decoupling, Transformations, Events

1 Introduction

In today's distributed software systems, vast amounts of data are generated and processed automatically, such as the tracking of goods by continuous updates called *event notifications*. Due to the high frequency of such data, distribution to clients must happen automatically based on client interests. This makes *implicit invocations* [32,39] (publish/subscribe – pub/sub [29]) the paradigm of

* 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 # 01IC10S01 and # 01IS12054. This work was performed within the LOEWE Priority Program Dynamo PLV (http://www.dynamo-plv.de) supported by the LOEWE research initiative of the state of Hesse/Germany. The authors assume responsibility for the content.

choice. In these systems, subscribers express their interests in data in the form of subscriptions. Broker nodes route matching notifications from publishers to the appropriate subscribers. Communication thus happens in an *n-to-m* fashion without direct references between communicating components. The federated software systems supporting these communications are loosely-coupled, highly heterogeneous and developed by many parties.

Heterogeneity is approaching. The need for such systems becomes apparent in today's economy, where large-scale software systems manage complex supply-chain networks. Cooperation happens across the globe between companies of different countries, cultures and structures. Software systems generate a huge amount of information that has to be distributed in business real-time among software components in a flexible way. For example, today's complex supply chains involve many companies world-wide; production strategies like *just-in-time production* have strongly increased the need for continuous low-latency flows of information between participants in a chain (e.g., monitoring transported goods) [13].

The continuing globalization of the economy[1], along with disruptive trends like the *Internet of Things* or *ubiquitous computing resources* promoted by the cloud paradigm and availability of event notification systems for clouds (e.g., Amazon Simple Notification Service [1]), will lead to more heterogeneous push-based distributed systems [20]. 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 [7]) further supports the trend.

Integration is challenging. The problem that arises in such heterogeneous environments are different data representations and data semantics of components. Local interpretations – *contexts* – differ based on geographical, cultural, legal, but also technical reasons. Programming languages differ in their notions of types, and even within a language, two sets of modules developed independently are not always easily integrated, e.g., due to *single* inheritance [12]. However, without correct interpretation of data, proper matching of *event objects* to subscriptions will fail as the anonymous n-to-m interaction between the components does not reveal intended bindings. Integration of information flows between publishers and subscribers is challenging as it must fulfill a number of requirements:

Expressiveness. Mediating between different unit systems (e.g., Fahrenheit and Celsius) requires value-based transformations. Many entities are encoded with several attributes, without 1:1 correspondances between types (e.g., Cartesian and Polar coordinates), and certain integrations might involve adding attributes (e.g., adding an attribute with a default or **null** value such as the state for a European surface mail address in US format).

[1] Our two ongoing research projects DynamoPLV(http://www.dynamo-plv.de) and EMERGENT(http://www.software-cluster.org) investigate such scenarios

Efficiency. Given the high rates at which event objects are published, mediation must take place on the fly with low latency.

Adaptability. The considered systems need to be able to accomodate joining and leaving of client components. Such changes can also engender new kinds of integrated data. Given the size of these systems, stopping them to add new components or also to modify existing integration rules is infeasible.

In addition, it should of course be easy for a programmer to express how integration has to happen. Existing approaches do not address all of these requirements to the full extent, as detailed in Section 7.

Transformation supports integration. In this paper we propose a framework for programming federated distributed software centered on object *transformations*. In our approach, we assume that each component (i.e., subscriber, publisher or pub/sub broker) resides in its own semantic context that involves a set of (abstract) parameters (e.g., country, programming language, development team) which governs how the component interprets event objects. We advocate a facility to define contexts including (1) *local types*, (2) fine-grained declarative *transformation rules* (*t-rules*) to specify the desired transformations between types, and (3) high-level *type mappings* expressing the programmer's intent and used to verify that the t-rule application yields consistent outcomes. Figure 1 shows an overview of our approach. Contexts can be extended at runtime and reused across components.

Our intermediary model avoids defining every possible transformation from any publisher to any subscriber, in a way similar to the Canonical Data Model in Enterprise Application Integration [22]. Thus we avoid $n \times m$ complexity of the transformation set and keep the set maintainable. This does not sacrifice adaptability however, since we do not require that the Canonical Data Model is established a-priori. Instead, it can be extended and modified at runtime by defining new contexts and compiling rule-sets on the fly. In fact, an intermediary model decouples publishers and subscribers further in that changes to a publisher's context (i.e. its transformation rules) do not affect its subscribers. Our model does not make assumptions on the specification language and is thus language-agnostic.

Contributions and roadmap. In this paper we

1. present an expressive and extensible model for federated software systems targeting mainstream programming languages used for such applications (e.g., C++, C#, Java) centered on *transformation rules* (t-rules);
2. introduce *contexts* as a reusable, higher-level abstractions of transformations, providing easy maintainability;
3. describe the implementation of our model in the ACTReSS system [19] based on the popular open-source ActiveMQ [37] event broker;

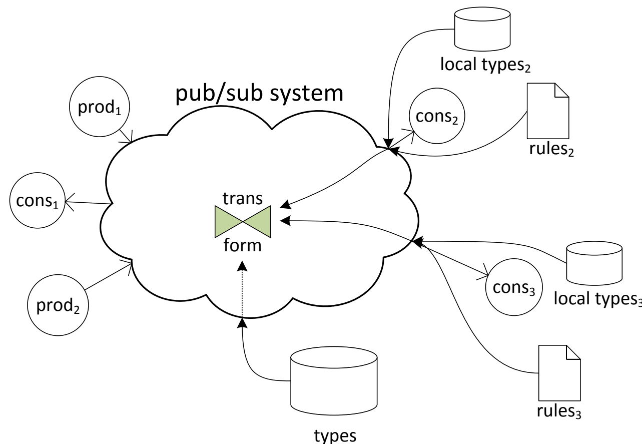


Fig. 1. Architectural overview. Producers ($prod_i$) send messages to consumers ($cons_j$) through a pub/sub system. Each client supplies a set of mapping rules between the global types and its local types (local types and rules omitted for some clients for presentation purposes).

4. evaluate our implementation. We demonstrate how extracting and consolidating transformation code improves efficiency compared to an equally adaptable and expressive approach without inherent support, i.e., based on reflection, and equal efficiency to manually coded transformations at client components; we also show that code quality, in particular upon adaptation, is much improved compared to manual coding or monolithic object transformations based on existing frameworks.

In a companion report [17] we formalize a subset of our model supporting only single subtyping and prove its soundness. An earlier implementation of our ACTrESS prototype was presented in a previous publication [19] without breaking down contexts into t-rules and type mappings – the main artifacts that a system software engineer must deal with – and detailing their syntax and semantics. Evaluation elided costs for extension or code quality improvements.

The remainder of this paper is organized as follows. Section 2 presents preliminary information. Section 3 presents t-rules through intuitive examples. Section 4 details contexts with particular focus on their relationships and type mappings. Section 5 presents the implementation of our model in ACTrESS. Section 6 evaluates its benefits. Section 7 discusses related work and Section 8 draws conclusions.

2 Preliminaries

We outline the notions of objects and transformations considered in this paper, using Figure 2 for illustration. Please note that for presentation purposes, the given example is small; our model allows for more complex transformations.

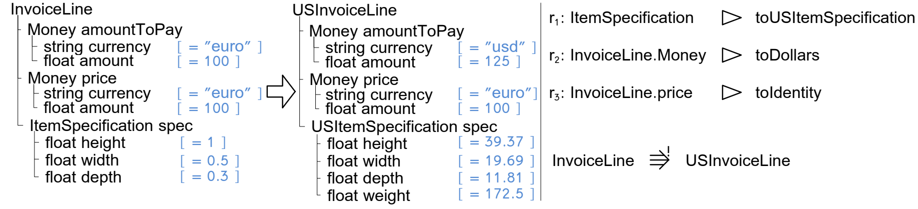


Fig. 2. Sample transformation. InvoiceLine and USInvoiceLine are context-specific interpretations of same objects. Note that currency is typically encoded explicitly, but units for other values are almost never encoded. Simplified transformation rules (t-rules) and type mappings are illustrated on the right.

2.1 Objects

We consider event objects in the general form of typed records of attributes. More precisely, such an object is an instance of a (complex) *type* ($\overline{z}$ denotes a sequence $z_1 \ldots z_n$):

Definition 1 (Type) *A type T is either a primitive type or a complex type declared as T* **extends** $T_1,\ldots,T_n\,[a_1 : T'_1, \ldots, a_w : T'_w]$. *$\overline{T}$ are super-types (complex) of T. T's attributes include those of all its super-types as well as $\overline{a}$.*

Thus, in a nested fashion, attributes of event objects can be objects. For example, InvoiceLine $[\overline{\delta}]$ would represent an event object of type InvoiceLine defined in Figure 2. The record $[\overline{\delta}]$ 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). We consider all transferred objects to be *values*, and thus our considerations also apply to other remote communication models with value semantics (references are usually built atop [5]). Note that an event object is an object but not every object is an event object.

We omit member functions/methods from types as our approach does not require those to be defined as part of types. The function $attrs(T)$ returns the attributes $\overline{a}$ of type T with their respective types $\overline{T}$. $T \preceq T'$ means that T is a subtype of T', i.e., T has been explicitly declared as a subtype of T', or one of the super-types of $T$is a subtype of T'. Our approach does not depend on the way in which attribute name clashes are handled.

2.2 Transformations: An Intuition

A very simple form of transformation of 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., meters or inches (cf. height, width and depth of InvoiceLine.spec in Figure 2). A similar case are conversions of primitive values between different architectures or platforms. On the other end of the spectrum of transformations, an object may be transformed in a way which affects

its internal structure, e.g., by merging multiple attributes, dropping attributes, and instantiating new ones (cf. weight of USInvoiceLine.spec in Figure 2). Any combination of these may be used to deal with versioning – by adding version numbers to type names and describing corresponding transformations.

There are different dimensions along which one can divide the space of object transformations. Section 7 further dissects this space in order to relate existing work to our proposed approach. The goal in our present work is to support (a) fine *granularity* – transformations on any attributes, at any nesting level, in objects; (b) strong *completeness* – function-based stateful transformations. (a) and (b) together yield the required expressiveness (see Section 1), while efficieny is supported by a decentralized application of transformations. These features as well as adaptability and ease of use are achieved by our design outlined in the following sections.

3 Transformation Rules

We introduce our approach to contextualization by starting from transformations which currently are dealt with in a very explicit manner by programmers. We follow the example of a logistics provider operating world-wide who has to communicate with customers from different countries.

3.1 Overview

Our model is centered around declarative *transformation rules* (*t*-rules) which minimize the necessary specifications and align well with programmers' mental models. These t-rules apply a given transformation function to attributes in event objects. The application of these transformations proceeds in a top-down fashion following the nested structure of event objects. Conceptually speaking, in example from Figure 2, imagine an object of type InvoiceLine being traversed attribute-wise, i.e., amountToPay, price, and finally spec, and any corresponding resolved t-rules being applied. If Money is a composite type, then the attributes of amountToPay are traversed recursively (depth) before proceeding with price (width). At any point in the transformation process, we thus consider one *path*:

Definition 2 (Fully qualified path) *A fully qualified path is a 2-tuple* $\langle T_0 \cdot ... \cdot T_n, a_1 \cdot ... \cdot a_n \rangle$ *such that* $\forall i \in [0..n-1]$, $attrs(T_i) = \langle ...a_{i+1}..., ...T'_{i+1}... \rangle$ *and* $T_{i+1} \preceq T'_{i+1}$.

A (fully qualified) path thus unambiguously and correctly denotes a given attribute within event objects. A prefix of a path (e.g., $\langle T_0 \cdot T_1, a_1 \rangle$ for $\langle T_0 \cdot T_1 \cdot T_2, a_1 \cdot a_2 \rangle$) is a path itself. Next we elaborate on how to describe t-rules and how they apply to paths.

3.2 Separating Patterns and Functions

The logistics provider from our example receives the specifications for the items to be transported from its customers. However, 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 calculating 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, the logistics provider sends the price along with these specifications to its customers. A US customer needs these properties in Imperial units (e.g., inches) rather than the logistics provider's 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;
```

t-rules are thus of the following form

Definition 3 (Transformation rule) *A transformation rule (t-rule) is of the form $p \triangleright f$ where p is a* pattern *delineating a set of attributes in event type(s) and f refers to a function.*

Functions are used to transform at any path which the pattern applies to, and are defined separately from t-rules. This separation between patterns and functions is key to expressiveness and ease of use. In an object-oriented programming language such as Java, functions are typically methods on transformed objects (e.g., $p \triangleright$ m with m an instance method in the type expected from p) or **static** methods (e.g., $p \triangleright$ C.m). 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 (...); }
```

We will unveil the details of patterns as we move on and present a precise definition after that.

3.3 Types and Nesting

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 event objects 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 attributes of type ItemSpecification, we introduce 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 rules with less specific ones. In the above example, CustomsDeclaration.ItemSpecification overrides ItemSpecification. 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 though, 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.

3.4 Attributes

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 respective local currencies. Thus we cannot treat every attribute of type Money the same way.

We consequently support references to *attributes* as another 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");
    // getRate() might depend on the current time and other state
    m.setAmount(m.getAmount() * getRate("euro'',''usd"));
  } else if (m.getCurrency() == "yen") {...}
  ...
  return m;
}
```

to *only* the amountToPay attribute of events of type InvoiceLine . 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;
```

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 reasoning behind this second form of precedence consists in prioritizing *instances over types* on otherwise comparable patterns, i.e., attribute-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 .

3.5 Subtyping

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 T' which is a subtype of a's declared type T ($T' \preceq_P T$). 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) ▷ ...;
```

Among a set of alternative patterns it seems natural to follow *subtyping level*, i.e, pick the one which refers to the most *derived* type at a point of comparison. For example we select the above rule over one with pattern InvoiceLine .spec(ItemSpecification),or InvoiceLine .spec for short, for an attribute weight of *dynamic* type WeightedItemSpec in an InvoiceLine . Note though that, unlike in dynamic method dispatching [10] this selection itself is not done dynamically; as we will elaborate on in Section 5, relevant t-rules are resolved before actually being applied.

In conclusion from the above, we can now proceed to more formally specifying the shape of *patterns:*

Definition 4 (Pattern) *A pattern $p = T.q_1.\ldots.q_n$ consists in a type reference T followed by a (possibly empty) sequence of* qualifiers*, and denotes attributes in event objects. A qualifier refers either to all attributes with a given type (* type qualifier T *) or to a given attribute "a" with a given type T (* attribute qualifier *$a(T)$).*

We only consider *valid* patterns. That is, for any prefix $T.q_1.\ldots.q_i.q_{i+1}$ in such a pattern, let T_i be the type of q_i:

- if q_{i+1} is a type qualifier T_{i+1} then T_i contains at least one attribute of type T_{i+1};

– if q_{i+1} is an attribute qualifier $a_{i+1}(T_{i+1})$ then T_i contains an attribute a_{i+1} of a super-type T'_{i+1} of T_{i+1}.

As already alluded to, we allow attribute qualifiers to be declared without type, e.g., $T.\overline{q}.a$ instead of $T.\overline{q}.a(T')$. In this case we adopt a's type according to its pattern prefix $T.\overline{q}$.

3.6 T-Rule Resolution

The three ideas on precedence (nesting level, instances over types, subtyping level) can conflict. For example a deeper nesting level of a type qualifier vs. an attribute qualifier.

We use the following priorities for competing t-rules. While P1-P3 summarize the precedences we introduced in Sections 3.3-3.5, P4-P6 specify how to resolve conflicts between these.

P1 Nesting level: A natural choice consists in considering nesting level as prioritizing measure among patterns (see Section 3.3). Deeper nesting levels translate to more detailed knowledge about data-structures and thus to more specific behavior. Thus a t-rule with the pattern $T_0.T_1$ will be chosen over one with pattern T_1 for attributes of type T_1 at a path rooted at a type T_0.

P2 Instances over types: Another intuitive choice (see Section 3.4) consists in giving attribute qualifiers priority over type qualifiers. Thus $T_0.a_1(T_1)$ is chosen over $T_0.T_1$ for attribute a_1 in an event of type T_0.

P3 Subtyping level: A third intuitive choice (see Section 3.5) is to consider for any otherwise equivalent patterns the ones which use qualifiers with the most derived types: with T'_1 a strict subtype of T_1, $T_0.a_1(T'_1)$ is chosen over $T_0.a_1(T_1)$.

P4 Subtyping order: In languages like C++, C#, or Java, in which many federated distributed systems are developed, a type can have *multiple* super-types. This leads to tie-breaking issues similar to those encountered for multiple inheritance (e.g., selection from multiple methods with equivalent signatures). For instance, there might be two patterns (a) $T_0.T'_1$ and (b) $T_0.T''_1$ which both apply to a path $\langle T_0 \cdot T_1, a_1 \rangle$ where T_1 is a subtype of both T'_1 and T''_1. Assuming that the subtyping level of T_1 is the same with respect to T'_1 and T''_1 (otherwise P3) takes over, we consider the *order* of subtyping declarations for breaking ties. For instance, with T_1 **extends** $T'_1, T''_1 \ldots$ (a) will be chosen.

P5 Instances over nesting level: We need to break ties between P1 and P2). Assume that we are transforming at a path $\langle T_0 \cdot \ldots \cdot T_3, a_1 \cdot a_2 \cdot a_3 \rangle$. Now consider two matching patterns (a) $T_0.a_1(T_1).T_3$ and (b) $T_0.T_1.T_2.T_3$. 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 T_3 immediately in a_1, T_3 is expressed for that prefix $T_0.a_1(T_1)$ which is more specific than the corresponding prefix $T_0.T_1$ in pattern (b), and thus (a) is prioritized.

P6 Subtypes over instances: We also need to break ties between P3 and P1, and between P3 and P2. Both ties are broken by favoring subtyping over instances

(P3 over P5): we consider first a qualifier's type and then only whether it refers to an attribute or a type. Thus when comparing a type qualifier T with an attribute qualifier $a(T')$ we prioritize the first if T is a strict subtype of T'; inversely the latter. If $T = T'$ we follow P2, i.e., favor the latter.

A formal characterization of t-rule resolution semantics and arguments for its type safety are the subject of [17].

4 Contexts

This section defines contexts and higher-level abstractions for transformations. Contexts allow for grouping a common set of t-rules and types together providing better abstraction and adaptability.

4.1 Local Types and Type Mappings

Components in the targeted applications execute in given contexts. Formally, such a context is defined as follows:

Definition 5 *A* context *is a 3-tuple* $(\overline{t}, \overline{u}, \overline{r})$ *where*

- $\overline{t}$ *is a set of type definitions (see Definition 1),*
- $\overline{u}$ *is a set of type mappings of form* $T \Rrightarrow^h T'$*, where* $h \in \{?, !\}$ *indicate transformations of incoming/outgoing objects respectively*
- $\overline{r}$ *is a set of t-rules (see Definition 3).*

The previous examples did not specify any signatures for the functions referenced by t-rules. In instantiations of our model in object-oriented programming languages, methods are used as such functions. In a language with overloading/overriding, a (partial) signature can be used to distinguish between different possibilities. In general, these functions need not return values of the same type as their formal arguments, which allows for type conversions. Such conversions are desired when different interacting components use different sets of types which can not be linked via subtyping due to practical restrictions on type systems [12]. To capture this, a context c includes local type definitions. Such a definition t introduces a new type T for c.

If the type of any attribute of a transformed event object changes, then the type of the entire event has to change. A context thus includes a set of high-level *type mappings* u, each of the form $T \Rrightarrow^h T'$ indicating that instances of event type T are mapped to event type T'. h denotes whether the transformation takes place upon received (?) or sent (!) event objects. Type mappings make the programmer's intent explicit and are used for type checking t-rules as described in the next section. The type mapping $\mathsf{InvoiceLine} \Rrightarrow^! \mathsf{USInvoiceLine}$ in Figure 2 for instance represents the mapping for the logistics provider to a US context. Contexts thus decouple *which* types have to be mapped to each other (mappings) from *how* this happens (t-rules), increasing flexibility.

4.2 Context Specialization

Contexts are arranged in a hierarchy following a notion of specialization which is similar to that of inheritance in common class-based programming languages. We elaborate on the meaning of inheritance and its relation to transformation of event objects in the following:

Inheritance. Inheritance of local types, mappings, and t-rules between a parent context such as c_0 in Figure 3 and its child context c_1 obey the following rules:

- *Local types* are straighforwardly inherited by c_1.
- *Type mappings* are inherited, but can be overridden by c_1. That is, if c_0 defines $T \Rrightarrow^h T'$ for some h, T, and T', then c_1 can redefine a mapping $T \Rrightarrow^h T''$ which overrides that of c_0. Overloading can also happen, even within a same context, through the subtype-sensitivity of mappings – for example, for two types T, T' such that T' is a subtype of T, a mapping $T' \Rrightarrow^? \dots$ takes precedence over $T \Rrightarrow^? \dots$ for instances of T'.
- *t-rules* are similarly inherited, unless overridden. Overriding here occurs when two t-rules have *identical* patterns, otherwise the patterns co-exist (with one taking precedence over the other based on the priorities listed in Section 3.6).

The *root context* c_0 in Figure 3 is considered to represent all reference types; put differently, all types of the root context are inherited by all other contexts. In a multi-language setup, this root context would typically include types defined in an independent declaration language, with one child context per supported programming language.

Transformation. Despite a possible "chain" of context specializations, transformations in our current model are always to and from the root context as shown in Figure 3. That is, events produced in a context (e.g., c_3) are *directly* transformed to the root context c_0, if needed in any different context; events in the root context are transformed *directly* to any other context (except the original one, e.g., c_3).

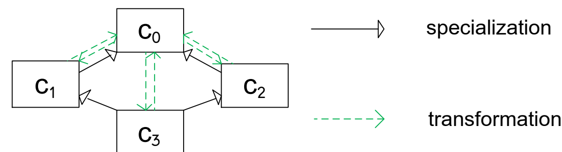


Fig. 3. Context specialization and transformation paths. Solid lines represent specialization. Dashed lines represent (optional) transformations

While in some cases the transformation could take place indirectly by following the inheritance relation (e.g., from c_0 to c_1 to c_3 in Figure 3), this would in

general raise many issues. For example, different paths could be possible (e.g., from c_0 to c_3 via c_1 or c_2), and one could expect that they produce consistent outcomes which is hard to assert.

Contexts may not always define mappings from all types T in the root context c_0 to the specific context (e.g., c_3), or from all types T' in the context c_3 to the root context. We however refrain from forcing the developer of such a context c_3 to define these mappings and transformations. After all, there may be no semantically sensible transformation. The absence of a mapping will be however noticed when compiling context c_3 and signaled as an *observation* message. We elaborate further on conformance as well as *warning* and *error* messages when discussing implementation issues shortly in Section 5.

4.3 Declaring Contexts

To remain independent of a specific programming language, we support context declarations in the widely-adopted XML (other languages are possible). Thus, developers do not need to learn new specification languages. To illustrate this, we give a brief intuition how a context for Java can be declared in XML:

```
<types>
  com.logistics.us.USInvoiceLine
  com.logistics.us.USItemSpecification
</types>
<mappings>
  <mapping from="com.logistics.eu.InvoiceLine"
        to="com.logistics.us.USInvoiceLine"
        dir="!" />
</mappings>
<rules>
  <rule pattern="ItemSpecification" function="toUSItemSpecification" />
  <rule pattern="InvoiceLine.Money" function="toDollars" />
  <rule pattern="InvoiceLine.price" function="toIdentity" />
</rules>
```

As the listing shows, all three elements of the context-tuple can easily be encoded in XML.

4.4 Practical Extensions

We provide several syntactic shortcuts for conveniently dealing with t-rules in our model. Noteworthy here are

- t-rules can be explicitly disabled upon inheritance in a child context c'. To that end, the developer can simply repeat the corresponding pattern, and use '-' in lieu of a function name. We are currently investigating labeling schemes so patterns do not need to be repeated.

– As showcased in Section 3.3, the `toIdentity` function, it is quite convenient to exclude certain attributes from a default transformation. In many cases, this can be simpler to do than enumerating transformations individually for all non-exempt attributes. For convenience we thus provide a polymorphic `toIdentity` function which simply returns its argument. An instance of this function in a given t-rule adopts its argument type as return type.

5 Implementation

This section presents our implementation of t-rules and contexts. We give details on rule resolution, error handling in case of invalid rules and how we improve on efficiency by generating static code.

5.1 ACTrESS

ACTrESS ("Automatic Context Transformation for Event-based Software Systems") implements our approach for the Java programming language[2]. ACTrESS is built on top of ActiveMQ [37], a fast, reliable JMS [8] broker. Our approach is implemented as a *plugin*. 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.

5.2 Rule Resolution and Validation

Our implementation uses a type system [17] to resolve relevant t-rules in a given context with respect to produced and consumed event types. That is, the type system performs the following tasks:

A. *Type-checking of individual t-rules.* For any t-rule $p \; \triangleright f$ in a given context, the type system first validates the pattern p (see Section 3.5), and then ensures that the formal argument of the function f indeed is a super-type of the expected type based on p (e.g., the expected type for a pattern $T_0.a_1(T_1)$ is T_1).
B. *Resolving t-rules.* Our type system identifies for any given event type T mapped in or out by a process all transformations for all *reachable* paths rooted at T, and retains these. This retained information is of the form $\langle T_0 \cdot \ldots \cdot T_n, a_1 \cdot \ldots \cdot a_n, f \rangle$, prompting the evaluation semantics to apply function f at the path $\langle T_0 \cdot \ldots \cdot T_n, a_1 \cdot \ldots \cdot a_n \rangle$ in any event of type T_0. These t-rules are resolved by starting from all subscribed and published types T_0, and exploring their attribute spaces recursively by following breadth (e.g., $\forall a_1$ s.t. a_1 is declared by T_0) and depth (e.g., $\forall \; a_2$ s.t. $a_1's$ type T_1 declares an

[2] More on ACTrESS can be found at http://www.dvs.tu-darmstadt.de/research/events/actress/

attribute a_2). To deal with subtyping, for a given path (e.g., $\langle T_0 \cdot ... \cdot T_i, a_1 \cdot ... \cdot a_i \rangle$) the *subtype space* is explored similarly in a recursive manner (e.g., $\forall T_i' \preceq T_i$). A reachable path implies that there is no transformation for any of its prefixes; exploration does not proceed further when a transformation is resolved, as the respective function is responsible for dealing with nested attributes.

C. *Type verification.* For any t-rule involving a function f to be applied at a given path, we verify whether the type returned by f abides to the type stipulated by the mapping for the corresponding event type. 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. Thus 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. 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.

This validation and resolution is performed at compilation, and extended at need at runtime, i.e., upon encountering new (sub)types. (Cyclic) recursions in types are handled in a way similar to iso-recursive types by *unfolding* upon *resolution* and *folding* upon *application*. That is, we halt exploration upon encountering recursion, and at a given path apply t-rules identified for prefixes of the path without the recursion. This implies that we do not support "recursion-sensitive" patterns such as $T_0.T_1.T_1$. We believe this would unnatural for programmers.

5.3 Errors and Safety

When t-rule resolution and compilation discovers invalid patterns (see Section 3.5), or failed type checks (see A. and C. above), it quits with corresponding error messages. Since resolution and compilation is done on a per-context basis, only the affected context will be unavailable (or remain unchanged if it already existed). The system will continue operating with all other contexts. We believe this is a better approach than permitting faulty t-rules and hoping that they do not trigger an runtime, or simply ignoring corresponding errors. An error in t-rule resolution is usually symptomatic of more profound inconsistencies.

Warnings are issued when a context contains several mappings with identical source (mapped) type or several t-rules with identical patterns; the last such mapping or t-rule is chosen respectively. Note that through the addition/discovery of a new subtype T' of an (attribute or event) type T no errors can be introduced, as the existing mappings and t-rules remain valid. That is, mappings remain trivially the same.There can not have been any "dormant" mapping specifically referring to T', otherwise the resolution process would have known that type (hence it's not new) and would have considered all applying t-rules. Similarly, a now active but previously disregarded t-rule must have referred to T' in its pattern already, leading again to a contradiction.

5.4 Code Generation

Besides avoiding repetitive t-rule resolution, our compilation approach has the advantage of being able to generate static code for performing transformations rather than using reflection mechanisms to dynamically invoke such functions/ methods. That is, our prototype generates a class containing transformation code after analyzing t-rules and notification types. This is also the reason why we proactively explore all subtypes and retain corresponding transformations; it avoids performing any kind of resolution at runtime. We will illustrate the efficiency benefits of our approach shortly in Section 6.

5.5 Annotations

To allow for intuitive and in-code declaration, our Java prototype supports various Java annotations. To define type mappings, the developer can use the **@MapsTo** annotation, supplying the class name of the mapped class. To specify that a class should be transformed with a specific function, developers may use **@TransformWith**.

These annotations are just another form of expressing t-rules. To simplify things for developers further, we allow annotating a class's attributes with units (e.g., **@Unit**(''USD'')). By analyzing the units given by developers and those that the notification service uses, our prototype is able to generate the appropriate t-rules. Thus, developers can simply express their *wish* towards the data.

Our supplied annotations are not as expressive as the full t-rules but cover many typical application scenarios. Developers can use annotations to quickly generate the majority of t-rules and then fine-tune the rule set.

6 Evaluation

We evaluated our approach and implementation with regards to performance and code quality. Our results illustrate that our approach is suited for pub/sub systems by providing efficiency and extensibility while being expressive.

6.1 Performance

First we substantiate the claim made earlier that native support for transformations is beneficial for performance, by showing that (a) it is much faster than an analogous library implementation based on reflection permitting equal expressiveness and extensibility, (b) it is much faster than Apache Camel, a popular general-purpose Enterprise Application Integration (EAI) [22] framework, (b) it is as effective as manually coded transformations in application components, and (d) the application of transformations closer to producers, enabled by our model, further improves performance. Furthermore, we show that the implementation of our model scales with the number of types and t-rules in the system.

We ran the ACTrESS broker, the workload generators and data collectors in a distributed environment. We compared five setups: in *content-based*, brokers just access event content for content-based routing but do not perform any transformations, assuming all parties agree upfront on types; *model-tx* transforms event objects according to our approach, while *reflect* uses Java reflection for transformation resolution and application. Setup *camel* uses Apache Camel for transformations to investigate the impact of using EAI frameworks. EAI frameworks share the idea of integration with our approach, but do not provide implementation details. (see Section 7 for details). We use compiled transformation classes for this approach, like those that our approach generates. Finally, in the *baseline* case brokers simply forward event objects without accessing their content, illustrating the smallest latency possible.

The SPECjms2007 standard benchmark specifies a workload where distribution centers, headquarters, and suppliers form a complex supply chain and interact via various inter-company event notifications [35]. We designed a workload following SPECjms2007. Event types are taken directly from the specification. Transformations include addresses, distances, and currency.

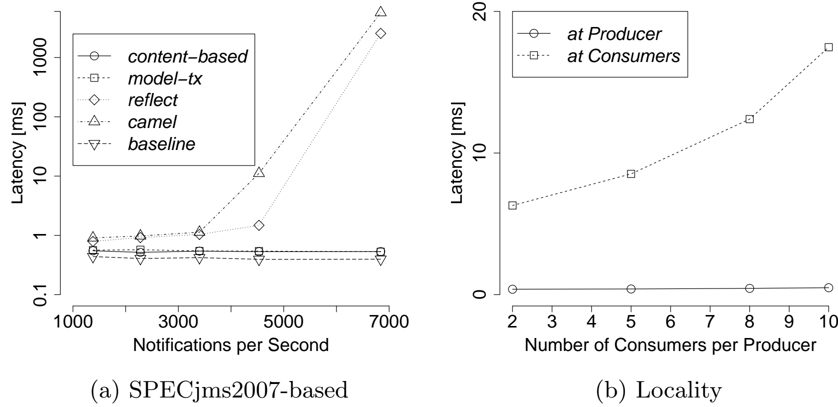
Figure 4a shows the latency for different notification rates. It is important to note that the ordinate has a logarithmic scale. As the figure shows, *reflect* adds significant latency overhead compared to our approach. This is an indicator for the increased computational effort of the reflection-based approach. Due to the increased effort, ActiveMQ cannot cope with higher event production rates; events are stalled, accounting for the steep latency increase. *Camel* performs even worse, because it is not integrated into the broker and thus additional notification marshalling and unmarshalling has to occur. This shows that even by requiring manual implementation of transformations, EAI frameworks also suffer from poor efficiency.

Scenarios *none* and *compile* achieve more than double the throughput than *reflect* and have no measurable difference, while *base* achieves even slightly more events per second. Because there is no measurable difference between *none* and *compile* and our setup already resembles the worst case where every event has to be transformed, we do not provide more details like the effect of the number of contexts.

Figure 4b illustrates the performance benefit of being able to transform closer to the producer, compared to doing it at each individual consumer. With a growing number of consumers per producer, the advantage grows. Thus, it is beneficial to be able to transform close to the producer, which our model enables.

6.2 T-rule Resolution and Code Generation Overhead

Our implementation generates transformation code from the set of rules and type mappings. While we believe that compared to the actual event notification rates, changes to the rule set, the type set or the mappings are rare, we are still interested in the overhead of this step. It is important that this step has acceptable overhead so that deployment and testing can be done quickly and – even more importantly – changes at runtime take as little time as possible.



(a) SPECjms2007-based (b) Locality

Fig. 4. Performance benefits

Figure 5 shows the time it takes to generate 1000 transformation classes for different sizes of the t-rule set and different selectivity of the rules. Selectivity means how many attributes of the type are actually affected by the rule. We used 1000 iterations to keep the variance low. Our implementation scales linearly with the number of t-rules and affected attributes. Every t-rule has to be checked because there might always be a more specific one at the end, and thus this is the optimal result. Similarly, for each affected attribute, we have to generate some code. Thus, a generator must have at least linear complexity.

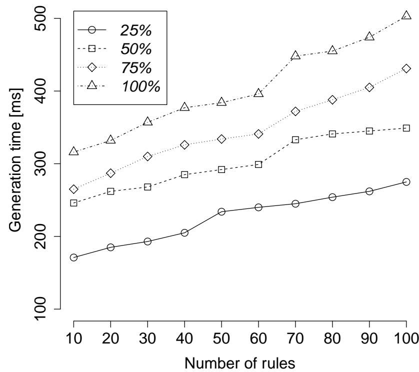


Fig. 5. Generation time

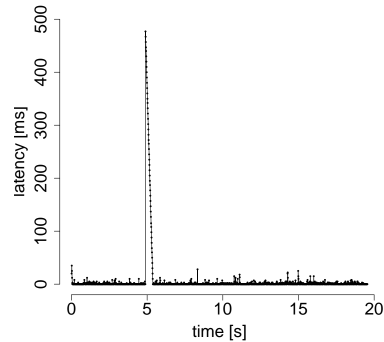


Fig. 6. Recompilation overhead

6.3 Ease of Use

Next, we demonstrate efficiency for the programmer by showing that our contexts lead to a considerably lower implementation effort than manual coding of transformations, and that existing transformations can be changed more easily. We use five typical event-based applications to compare the required lines of code and ease of changes.

Kemerer [23] analyzed over 30 software complexity metrics and concluded, that "a number of the more complex metrics may be essentially measuring the size of the program or other component under investigation, and therefore may provide little additional information". We thus see lines of code as a valid indicator for code complexity and maintenance effort and used it for our comparison.

To give a brief comparison between the two approaches, consider the example to transform attributes of type Address. Our approach needs just one rule to transform every occurrence of an attribute of type Address. When coding transformations manually, the developer has to write a dedicated **if-else** branch for every event type with an address attribute (directly or indirectly). Inside, several lines of code for attribute extraction, object creation and transformation are needed.

For our comparison, we use the event types specified by various applications: The SPECjms2007 benchmark introduced in Section 6.1. Transformations include address translations between different regional formats and changing product descriptions (in orders, invoices, etc.) to conform to each site's expectations.

The MARKETCETERA[3] automated trading platform defines various event types capturing stock ticker quotes and allows for elaborate automated trading. The transformations on this platform perform currency conversions, timestamp formatting changes and renaming of some indicators, assuming differing terminology on the consumers.

The HTM traffic management system handles data from sensors and cameras along streets and highways, monitoring road, traffic and weather conditions [34]. Such systems are used by many large cities. Transformations include coordinate translation, timestamp format changes and unit conversion.

DRADEL is an application environment for modeling and analyzing distributed architectures, including code generation [25]. Since it runs on top of a message-oriented middleware, multi-user support is possible. In such a setting however, path references and line numbers need to be adapted to each platform. Thus, operational events of DRADEL need to be transformed accordingly.

The Emergency Response System (ERS) is a distributed application running on multiple mobile devices and helps organizing human resources during natural disasters [31]. Its events often refer to geographical regions, which need transformation between individual users to adapt to their specific format.

Table 1 compares the lines of code needed to *specify* transformations by manual coding and by our approach, showing a clear benefit for the latter. The numbers indicate the effort necessary to specify the transformations of one client that

[3] http://www.marketcetera.com

Table 1. Code complexity comparison

	manual			*ACTrESS*		
	specify	*change*	*extend*	*specify*	*change*	*extend*
SPECjms2007	167	15	92	22	2	1
marketcetera	108	9	37	16	2	1
HTM	237	16	133	21	2	1
DRADEL	417	16	139	30	3	2
ERS	351	12	117	21	3	2

needs to transform events. We did not count lines of code that can be generated by standard IDEs. For additional clients needing different transformations, the effort has to be made again, multiplying our benefits. Furthermore, the *change* columns show the number of *individual places* in the code that needed to be changed when a certain type (e.g., Address) should be changed into a different format.

Contrary to intuition, EAI frameworks like Apache Camel do not reduce complexity at this step. Although supporting event transformations architecturally, transformations still need to be coded manually, resulting in above depicted effort.

6.4 Extensibility

Changes can roughly occur in two ways: transformation functions need to be changed or new types are introduced. In case of changed transformation functions, one can simply adapt them (e.g., change toUSAddress). Adding new types is more complicated. Suppose we want to add a sensor to the traffic management system to detect oil on the road (raising an Oil event). Every sensor event in the system has an attribute SensorMetadata which has an attribute of type Location. In the manual approach, it is thus not immediately apparent that adding the Oil event requires a new piece of transformation code. A developer will have to analyze the existing transformation code and realize that locations are transformed and then write the new code. With our approach, only the mapping has to be defined. The *extend* column in Table 1 illustrates this by giving the required number of lines of code for the necessary analysis.

In case of changes at runtime, ACTrESS dynamically recompiles the generated classes when changes occur. Figure 6 illustrates the impact on performance. It shows that there is a brief increase in latency for the recompilation, after which the system performs as before. This demonstrates that our implementation can handle changes at runtime without significant impact on performance.

7 Related Work

In this section, we divide the space of object transformations along different dimensions and relate existing work to our proposed approach along these dimensions.

7.1 Transformation Space

As mentioned there are different dimensions along which one can divide the space of object transformations. Among these, without attempting to be exhaustive but to cover the main related work, we can consider *granularity*, *completeness* and *topology*. The possibilities for *granularity* for instance include

- G1 Monolithic object transformations. Objects of given event types are transformed as a whole.
- G2 Attribute-wise transformations. Event objects are transformed attribute-wise, with a 1-1 mapping of attributes.
- G3 Nested attribute-wise transformation. With objects containing attributes that are objects, one can allow attribute-wise transformation of such nested objects.
- G4 Path-based transformation. Transformations can be expressed on any attributes, at any nesting level, in objects.

In terms of the actual transformation, there are also different levels of computational *completeness* that one can imagine:

- C1 Type or meta-data transformation. Objects retain their actual state, but they are converted to other types. This includes also traditional subtype subsumption, where simply a subset of the attributes are retained when accessing an object via a super-type.
- C2 Lookup-based transformation. An object is substituted by another one based on a lookup in a static or dynamic data-structure. Such objects correspond to discrete values.
- C3 Function-based transformation. A function is invoked with an object and can perform any computations to construct a substitute object.
- C4 Function-based stateful transformation. Same as above, except that the function can also persist state in variables.

There are also different places in the *topology* of a distributed application for transformation application, e.g.,

- T1 Peer-based transformation. Every application component or process performs its own transformations on incoming — maybe also outgoing — event objects.
- T2 Distributed transformation. A distributed middleware system performs transformations on conveyed event objects, through a dedicated server or component.
- T3 Decentralized transformation. Here a distributed middleware performs transformations without relying on a centralized component.

Our solution presented supports the highest level for any of these criteria, i.e., G4, C4, and T3.

7.2 Existing Work

In database integration, data from one database is transformed to adhere to the schema of another database [9,30] (T2). Anonymity in a federated pub/sub-based system prevents *schema integration* used by these approaches. A subscriber does not know who produced an event it receives and thus not the schema it follows.

Chung [14] argues that it is infeasible to decide upon a database for a whole organization, proposing DATAPLEX as a middleware layer implemented as centralized data mediation component (T2) to allow uniform access to databases.

Cluet et al. [16] 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, where subscribers may not know the origin (communication endpoint) of data. Cilia et al. [15] propose a solution to deal with heterogeneous data sources in pub/sub systems using a self-describing model. Neither of the above however verifies typing of transformations. Dozer [3] supports mapping of data objects between Java Beans. Dozer supports expressive and complex mappings; conversion resolution and execution occur via Java Reflection at runtime, limiting performance and safety.

Foster et al. [18] present a bi-directional tree transformation approach. Their transformation functions allow to mediate between different views of same data where updates are applied backwards to the original data (C3). In contrast, our approach purposely supports uni-directional and non-deterministic transformations. Every subscriber gets its own copy of an event and thus events are not meant to be shared like documents.

Several authors propose structural subtyping to decouple components (G3, C1), which has been promoted by several research programming languages (e.g., Lingua Franca [27], Accute [36]). Whiteoak [21] extends Java with structural conformance similarly to *compound types* [12].

HydroJ [24] extends Java with relaxed conformance on nested semi-structured events exchanged between processes. Similarly, most publish/subscribe systems follow the model described in [29] which promotes the use of hash maps to convey events in the form of $\langle key, value \rangle$ pairs. This places all burden on programmers as these need to manually inspect, marshal/unmarshal and transform event objects at generation and reception. With Hashtypes [36] type representations are hashed, including "contents" of corresponding instances, function signatures in modules, etc. Hashes are propagated with objects. Given the focus on point-to-point and not implicit communication, transformations are applied at end components (T1). None of these approaches support value-based transformations (C1).

Platforms like Sun RPC [38], OMG's CORBA [28], or Web Services [11] only mediate between *encodings* of values (e.g., little vs. big endian).

Java Internationalization (JI) [4] provides support for context-specific interpretation of precise data types (e.g., strings in different character sets, times in different zones). JI also supports automatic translation of character strings between different natural languages based on dictionaries (C2). JI furthermore in-

cludes a framework which allows application-specific types (*resource bundles*) to be interpreted differently across contexts (*locales*). In combination with Java Remote Method Invocations (RMI) [5] or other remote communication paradigms, JI can hence be used as a foundation to address similar problems as studied herein. However, the JI framework consists merely in an API, while the present work aims at providing intuitive and safe mechanisms for *implementing* such an API. .NET Internationalization [6] provides analogous functionalities to JI for the .NET platform. Similarly, design and architectural patterns such as *adapters* [26] provide a locus between application components to perform transformations but do not provide support for actually implementing them.

Enterprise Application Integration (EAI) specifies *message transformations* [22] to deal with heterogeneity. However, EAI just specifies a pattern (in fact, they can be seen as a more detailed definition of adapters), without any suggestions regarding its implementation. EAI Frameworks like Apache Camel [2] thus support message transformations, but provide merely an API with little support for implementing it. Transformations take place on the entire message bodies (`G1`) and there is no facility for a *canonical data model*, leaving its design to the programmer. As demonstrated, efficiency is poor. Microsoft BizTalk [33] provides support for transformation with XSLT and *orchestration*s. However, mappings are static, take place on entire message bodies (`G1`) and new producers or consumers have to be added explicitly. Other EAI frameworks expose similar limitations.

8 Conclusions

We have introduced a foundational model for interoperability in federated distributed software based on transformations. Our model is expressive in that it supports the whole spectrum of transformations including enrichment of events and allows other type conformance models to be implemented atop. As we have demonstrated it is easy to use by supporting fine-grained expression of transformations as opposed to monolithic ones, and it allows for extensibility at runtime, while at the same time showing being efficient, causing no measurable overhead on an underlying content-based pub/sub system. Last but not least, our approach is safe, by promoting clear semantics for transformations, whose application is verified and determined first and enforced at runtime.

We are currently working on supporting clients in other languages, as well as on an implementation of type versioning on top of our model. We are also investigating extensions to our model including nested transformations. These will allow functions used for transformations to explicitly re-invoke the transformation process in order to avoid invoking or repeating transformation functions for nested attributes. Care must be taken here to not increase expressiveness at the cost of simplicity. Last but not least, we are working on optimal placement of transformation operations in decentralized publish/subscribe networks.

References

1. Amazon Simple Notification Service, http://aws.amazon.com/sns/
2. Apache Camel Type Convert, http://camel.apache.org/type-converter.html
3. Dozer, http://dozer.sourceforge.net
4. Java Internationaliation, http://java.sun.com/javase/technologies/core/basic/intl/
5. Java Remote Method Invocation, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html
6. .NET Internationalization, http://msdn.microsoft.com/en-us/goglobal/bb688096.aspx
7. Project Kafka, http://engineering.linkedin.com/tags/kafka
8. Java Message Service - Specification, version 1.1. Tech. rep., Oracle Co., http://www.oracle.com/technetwork/java/jms/index.html (2008)
9. Adair, J., Coyle Jr, D., Grafe, R., Lindsay, B., Reinsch, R., Resch, R., Selinger, P., Zimowski, M.: Heterogenous Database Communication System in Which Communicating Systems Identify Themselves and Convert any Requests/Responses Into Their own Data Format. US Patent Number 5,416,917 (1995)
10. Allen, E., Hilburn, J., Kilpatrick, S., Luchangco, V., Ryu, S., Chase, D., Steele, G.: Type Checking Modular Multiple Dispatch With Parametric Polymorphism and Multiple Inheritance. In: OOPSLA 2011
11. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services. Springer Verlag (2003)
12. Büchi, M., Weck, W.: Compound Types for Java. In: OOPSLA 1998
13. Buchmann, A., Pfohl, H.C., Appel, S., Freudenreich, T., Frischbier, S., Petrov, I., Zuber, C.: Event-Driven Services: Integrating Production, Logistics and Transportation. In: SOC-LOG 2010
14. Chung, C.W.: DATAPLEX: an Access to Heterogeneous Distributed Databases. CACM 33, 70–80 (1990)
15. Cilia, M., Antollini, M., Bornhövd, C., Buchmann, A.: Dealing with Heterogeneous Data in Pub/Sub Systems: The Concept-Based Approach. In: DEBS 2004
16. Cluet, S., Delobel, C., Siméon, J., Smaga, K.: Your Mediators Need Data Conversion! In: SIGMOD 1998
17. Eugster, P., Freudenreich, T., Frischbier, S., Appel, S., Buchmann, A.: Sound Transformations for Message Passing Systems. Tech. rep., http://www.dvs.tu-darmstadt.de/publications/pdf/transsem.pdf (2012)
18. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for Bi-directional Tree Transformations: a Linguistic Approach to the View Update Problem. In: POPL 2005
19. Freudenreich, T., Frischbier, S., Appel, S., Buchmann, A.: ACTrESS - Automatic Context Transformation in Event-Based Software Systems. In: DEBS 2012
20. Frischbier, S., Michael , G., Mayer D., Roth, A., Webel, C.: Emergence as Competitive Advantage - Engineering Tomorrowś Enterprise Software Systems. In: ICEIS 2012
21. Gil, J., Maman, I.: Whiteoak: Introducing Structural Typing Into Java. In: OOPSLA 2008
22. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Professional (2004)
23. Kemerer, C.: Software Complexity and Software Maintenance: A Survey of Empirical Research. Annals of Software Engineering 1, 1–22 (1995)

24. Lee, K., LaMarca, A., Chambers, C.: HydroJ: Object-oriented Pattern Matching for Evolvable Distributed Systems. In: OOPSLA 2003
25. Medvidovic, N., Dashofy, E., Taylor, R.: The Role of Middleware in Architecture-based Software Development. International Journal of Software Engineering and Knowledge Engineering 13(04), 367–393 (2003)
26. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a Taxonomy of Software Connectors. In: ICSE 2000
27. Muckelbauer, P.A., Russo, V.F.: Lingua Franca: An IDL for Structural Subtyping Distributed Object Systems. In: COOTS 1995
28. Object Management Group: CORBA, http://www.corba.org
29. Oki, B., Pfluegl, M., Siegel, A., Skeen, D.: The Information Bus - An Architecture for Extensible Distributed Systems. In: SOSP 1993
30. Parent, C., Spaccapietra, S.: Issues and Approaches of Database Integration. Commun. ACM 41, 166–178 (1998)
31. Popescu, D., Garcia, J., Bierhoff, K., Medvidovic, N.: Impact Analysis for Distributed Event-based Systems. In: DEBS 2012
32. Reiss, S.P.: Connecting Tools Using Message Passing in the Field Environment. IEEE Software 7(4), 57–66 (1990)
33. Rosanova, D.: Microsoft BizTalk Server 2010 Patterns. Packt Pub Limited (2011)
34. 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
35. Sachs, K., Kounev, S., Bacon, J., Buchmann, A.: Performance Evaluation of Message-oriented Middleware using the SPECjms2007 Benchmark. Performance Evaluation 66(8), 410–434 (Aug 2009)
36. Sewell, P., Leifer, J.J., Wansbrough, K., Nardelli, F.Z., Allen-Williams, M., Habouzit, P., Vafeiadis, V.: Acute: High-level Programming Language Design for Distributed Computation. J. Funct. Program. 17(4-5), 547–612 (2007)
37. Snyder, B., Bosanac, D., Davies, R.: ActiveMQ in Action. Manning Publications Co. (2011)
38. Srinivasan, R.: RPC: Remote Procedure Call Protocol Specification Version 2 (1995)
39. Sullivan, K., Notkin, D.: Reconciling Environment Integration and Software Evolution. ACM Trans. Softw. Eng. Methodol. 1(3), 229–268 (Jul 1992)