Facilitating Re-Use by Design: A Filtering, Transformation, and Selection Architecture for Robotic Software Systems

Ingo Lütkebohle, Jan Schaefer and Sebastian Wrede

Abstract—Despite a number of contenders, no single robotics software framework has emerged as the standard, seriously impairing reuse at the component level. We propose a general pattern that structures components into sub-parts of lower granularity arranged in a transformation graph, thus making them more amenable to reuse. The graph representation supports both efficient execution and external configurability, to further reduce coupling and increase reuse. A framework-independent library is presented that realizes the pattern and provides a number of building blocks to easily integrate with existing software. We validate the approach using a case study from a current research system.

I. INTRODUCTION

Nowadays, a large number of software architectures are available that offer concepts and tools for the development of complex robotic systems [1], [2], [3]. Until now, however, no single architecture has become accepted by the community as a gold standard [4], [5], likely because of the diverse applications and research scenarios in advanced robotics. This has serious consequences for component reuse, which is often limited due to architectural constraints, such as the requirements imposed by different component interfaces and models [6].

Therefore, a key challenge in software engineering for robotics is to identify commonalities which could improve reuse of components across different frameworks. In this contribution, we aim squarely at this problem and identify a component structuring pattern that is re-usable across architectures and provide explicit support for reuse in two particularly important areas: Component packaging and pre-/post-processing.

The issue of how functionality is packaged has long been a concern in software architecture research. For example, in their seminal work on architecture, Shaw et al. discuss that various architectures are characterized mainly by packaging, e.g. as request-response, pipes-and-filters or blackboard architectures [7]. Garlan et al. emphasized that mismatches in packaging and other architectural issues such as requirements on framework components seriously hamper reusability [6]. In this paper, we also describe how packaging issues can occur even within a single system (cf. section II-C).

Packaging has sometimes been addressed by the creation of frameworks where all components fit tightly together. However, as mentioned already, there are many frameworks and transfer of functionality between them is even more difficult. Therefore, in this contribution we take an orthogonal approach and propose a software library designed to be used across frameworks.

Besides packaging, a second issue is the frequently very high amount of pre- and postprocessing, both regarding data transformations such as data fusion and selection and regarding control flow, such as synchronous vs. asynchronous dispatching. Many of these functions implemented in similar ways were found in the analysis of different robotics components. However, the corresponding code is often intertwined with other, problem-specific code. This “mashup” embeds system- and architecture specific assumptions deep in domain-independent code. Hence, a secondary, and complementary task is the development of a generic pre- and postprocessing infrastructure, to support the separation of reusable processing from problem-specific code.

The software concepts presented in this contribution address packaging and the pre-/post-processing tasks in a coherent approach that poses itself only minimal constraints on the environment in which it is applied. Our initial implementation of these concepts, the so-called Filtering, Transformation and Selection Library (FTS), consists of an externally configurable filter-graph structure that maps transformation and packaging plugins to asynchronous handler routines within software components. Furthermore, the library already provides a number of re-usable building blocks that are used as plugins within the filter graph. Finally, an asynchronous handler interface and a multi-threaded dispatching strategy has been developed in order to actually invoke the selected handlers within the software components.

While we have prototyped this library in the context of an event-based architecture [8], it is independently usable. Furthermore, we consider it as one implementation of a more general software pattern for components that is independent of any specific implementation.

In the remainder of this paper, we first shortly familiarize the reader with a current robotics system that serves as a case study for the presented concepts (cf. section II). Based on this example, we introduce the concepts behind the FTS approach (section III), then revisit the example system again in section IV, highlighting the changes that are due to our approach. Prior to the conclusion and outlook (section VI), we discuss related work in the software engineering and the robotics community (section V).
II. CASE STUDY

In the following, we will describe an exemplary scenario before and after application of the proposed approach, to demonstrate its effect. The scenario has been taken from a currently ongoing research project [9].

While a single case study cannot encompass the wide variety of robotic systems, our approach is based on the observation that many, if not all, systems exhibit similar component interaction flows. In other words, they exhibit a generalizable pattern in their interaction with the rest of the system. It is exactly this flow that the proposed approach is designed to capture and as a result, we expect that it applies to many other systems.

A. The Curious Robot Scenario

The “Curious Robot” is a human-robot-interaction (HRI) learning scenario with a twist: Instead of the human having to do all the demonstrating, the robot engages the human, mixed-initiative style. It analyzes its environment for interesting objects and asks the human about their labels, how to grasp them and so on. See figure 1 for a photo of the learning setup.

Besides its novel interaction style, the system is relatively standard and shares many attributes with other systems. It consists out of interacting components that work together to realize the observable functionality in a behavior-based architecture. Figure 2 shows a subset of the system’s architecture.

From these components, we have selected the so-called InterestRater (IR) for further analysis. It determines which visual region is the most interesting at the moment and also what the robot would like to do next with that region. Put more generally, it performs sensory input consolidation, target selection and also proposes an action from the list of possible ones. We believe this task to be a very general one that is present in almost all robotic systems and thus a good candidate for further study. Figure 3 shows an activity diagram of the IR.

B. Preliminary Analysis

From figure 3 we can already surmise that most of the IR’s functionality is actually reusable code. Firstly, acquisition and fusion of sensory information is similar for all consumers and can be generalized.

Additionally, because data acquisition communicates with other components, it is also closely coupled with the communications middleware. As such, reconfigurability of this layer would drastically reduce coupling.

The next processing step, selection of the most salient region, is the core of the component and depends not only on the input data but also on the components internal state. Hence, we do not consider it a candidate for reuse.

The select step, which determines what behavior to initiate, may be a somewhat surprising candidate for reuse because it requires knowledge of the available behaviors. However, we consider that the required information may be specified in the manner of a condition based on the input data. If this is done, the selection step may be performed by looking for the best match and passing that on to dispatching, without knowledge of the actual functionality.
C. Component Communication Mismatches

One issue of particular interest that arises in our case study is a mismatch in communication style within the architecture. Compare figure 2 for the different connector interfaces used.

One the one side, the visual analysis components provide their results continuously, as soon as they are available, in a publish/subscribe fashion. We believe this is fairly typical for bottom-up processing. In fact, it may even be a requirement for some components, due to safety issues in dynamic environments, which require immediate notification.

One the other side, interaction with the dialog manager and manipulator control uses the request-response style. Again, we believe this is fairly natural: For example, when the human partner answers a question, the answer is often only meaningful in the context of the question. This fact is reflected in the component communication style.

Such mismatches are known to impede reuse (compare [6]) and in this case, the difficulty is exacerbated by the fact that sensory input and human interaction occur at vastly different rates. In effect, the IR has to decide what to do with input data that arrives while there is still a request outstanding.

Depending on the operating context, several solutions may be practical. A simple one is to follow the limiting external component and act only when it is ready. Additional data from the input side is then only processed at those times, for example, with a use-most-recent-data policy. For our discussion, the relevant insight is that such a solution requires some form of buffering on the input side.

Alternatively, the IR component could always be active and generate one request for every input item. This simply puts the burden on the receiving component, which has to either queue or reject the inputs that it cannot process at the moment. We conclude that some form of adaptation would always be required.

The exact choice of adaptation depends on the system and it would be a big effort for any single component to hard-code a solution for all cases. Therefore, we conclude that reconfigurability of the adaptation component is a requirement.

III. TDL CONCEPTS

As the primary motivation behind this contribution was to provide a conceptual framework that increases the reusability of software components like the IR described previously, our initial step in this direction was to distinguish the reusable parts in typical information flow and separate them from the component-specific functionality. Compare figure 3 for an overview where the reusable parts have been boxed. As shown in the figure, we have divided the potentially reusable parts into four functionally distinct areas:

- **Filter**: the first identifiable set of tasks many components are executing at different places in their interaction flow is to filter incoming data against specific conditions.
- **Transform**: components in robotics applications regularly perform largely sequential transformations on their input or output data prior to processing, e.g., coordinate transformations.
- **Select**: selection statements usually choose between alternative pathways for data, often decided by some combination of data-checks and (optionally) internal state.
- **Dispatch**: as many components must react immediately to incoming stimuli, the – potentially asynchronous and multi-threaded – scheduling and dispatching of requests to method handlers encapsulating algorithmic processing is a critical function in robotics architectures.

Based on the identification of these four areas and through the analysis of several examples like the one presented in the previous section, we decided for a data-flow approach based on a generalized directed-acyclic-graph structure as the central abstraction to facilitate re-use. The so-called transformation graph allows to map building blocks of functionality into a processing model that is explicit and flexible, hence enabling re-use along all introduced dimensions.

Figure 4 depicts an exemplary configuration of the abstract elements in a transformation graph. While the nodes within the transformation graph are representing functions that can be classified into one of the four types introduced previously and detailed later on, the edges represent the possible (local) data-flow between different nodes. As a general policy, data is processed from left to right or from sources to sinks within exemplars of transformation graphs.

By design we consider every type of input, e.g., sensor readings, as events [10] which are submitted in the form of event notifications to the graph structure via event sources. Sources may be attached locally, in-process, or via remote access to external sensors. Event sources themselves are considered as filters as they deliver data selectively from different middleware drivers or programming APIs. The only limitation to event sources is their obligation to deliver events in a push-style semantics due to the acyclic nature of the transformation graph.

Subsequently, event notifications are submitted to the in-bound pre-processing part of the transformation graph consisting of re-usable filtering, transformation, or selection functions. A notification that reaches a selector may eventually lead to an invocation of an event handler function. These handlers are dynamically bound to the domain-specific processing code within a software component and are invoked by the dispatching functions.
Conversely, results of handler invocations are submitted to the post-processing part of the transformation graph consisting again of the three aforementioned types of nodes. After post-processing, the notifications are consumed by event sinks considered as filter nodes that selectively deliver data to local or remote interfaces like other components, actuator or sensors.

After this introduction into the general processing flow that we propose for an FTS architecture in robotics, let us now consider the actual re-useable elements and functions of the transformation graph and their specifics in some greater detail.

### A. Filter

Filters allow to narrow down the received set of notifications to a limited subset that is forwarded to its successors. Filters are usually stateless boolean-valued functions operating on a single notification. By applying a test on specific properties of the incoming notifications let alone whether this is based on the whole content, certain attributes or an expected sequence [10], they usually return true or false representing the success of the filtering operation.

Within the proposed architecture, a filter returns upon a successful match against its specified condition a reference to the unchanged event notification, which is immediately forwarded to its successor nodes in the transformation graph. In contrast, if a filter does not match a notification, the flow of this notification in the transformation graph is stopped and it is discarded if no other graph node references it.

Combinations of filters allow to observe specific events that occurred in a system. Consider for example that the human’s face was detected in front of the robot’s cameras with a certain probability. In this example, the filter graph would possibly feature a filter matching representations of faces, e.g., a type filter as the case in our library implementation, itself connected to a reliability filter in order to discard notifications containing detections with low probability.

### B. Transform

In the FTS architecture, the notion of a generic transforming function differs significantly from the concept of a filter. It is usually stateful and applies a transformation to the event notification it receives as an input.

In order to allow for parallel processing, we apply a copy-on-write strategy, where transform nodes are usually provided with indistinguishable references to event notifications. Upon modification, a deep copy is created to prevent the changes becoming visible in other threads and independent processing steps.

A transforming function implementation guarantees that its invocation provides a new result to its successors. If a transformation cannot be applied, this is considered as an error that is reported via an exception mechanism to the component that instantiated the transformation graph.

The ability to re-structure event messages allow to encapsulate integration functionality like coordinate transformations in re-usable software components that can neatly be integrated into the FTS processing model.

### C. Select

The paths of the graph may have interdependencies such that if one path is chosen, no other should be executed. A typical example of this is an execute-first policy, where only the first matching path is followed. Theoretically, this is equivalent to several filters with mutually exclusive patterns. However, we have chosen to identify as a first-class citizen to enhance the clarity of the resulting graph structure: When a node is labeled as a selector, it is immediately clear that only one of its outgoing edges will be followed.

Conceptually, a selector contains one pattern per outgoing edge. In the InterestRater example, this node type is used to determine which of the possible behaviors to start.

### D. Dispatch

While theoretically all node types can be treated equally in the graph, the event handler supplied by the developer often has different characteristics than the library-supplied transformation blocks. For one thing, developers are not expected to impose a limit on execution time. More importantly, they may want to use either synchronous or asynchronous invocation, depending on what fits their functionality better.

Scheduling concurrent executions according to their invocation mode is implemented by the dispatch building block. Synchronous invocations are handled through the standard java.util.concurrent ExecutorService and processing continues once the call is completed. Asynchronous invocations are realized through blocking, size-limited queues where the objects in the queue provide the input data and can also accept the output data. On provision of output data, processing in the graph continues.

As an interim conclusion, table I gives an overview of some instantiations of the aforementioned concepts, underlining that the proposed architecture improves not only the re-usability of the individual components, but also leads to the development of new building blocks of reusable code. Nevertheless, let us now turn to the example and see how the application of the aforementioned principles impacts on the structure of the described component.

### IV. CASE STUDY, REVISITED

Based on the TDL concepts, we have refactored the IR component into several, largely reusable sub-components. The only specific functionality now resides in three independent sub-components, which implement the concrete behaviors executed by the robot in response to inputs. All other aspects, such as data fusion, selection and dispatching have been moved to reusable blocks. In the following, we will describe these, with a particular emphasis on their reconfigurability.

### A. Data Transformations

To achieve reconfigurable data transformations, a generic data representation is required and to this end, attribute-value trees (AVT) or equivalent forms are often used, because they afford a name-based, extensible structure. We have chosen the XML Infoset [11], which is a well-defined AVT with a
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPath</td>
<td>Content-based matching with XPath expressions.</td>
<td>filter</td>
</tr>
<tr>
<td>Reliability</td>
<td>Evaluates notifications against a certain probability threshold.</td>
<td>filter</td>
</tr>
<tr>
<td>Identity</td>
<td>Matches on unique identity information.</td>
<td>filter</td>
</tr>
<tr>
<td>Scope</td>
<td>Reduces the visibility of events by introducing scopes.</td>
<td>filter</td>
</tr>
<tr>
<td>Type</td>
<td>Matches on event types and sub-types defined in the event model.</td>
<td>filter</td>
</tr>
<tr>
<td>Frequency</td>
<td>Filter that outputs only every n-th received notification.</td>
<td>filter</td>
</tr>
<tr>
<td>XQuery</td>
<td>Generic message transformation using XQuery.</td>
<td>transform</td>
</tr>
<tr>
<td>Compacting</td>
<td>Compares novelty of received notifications against previous ones.</td>
<td>transform</td>
</tr>
<tr>
<td>Select-First</td>
<td>Evaluates a number of conditions and selects the first that matches.</td>
<td>select</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Filter</th>
<th>Transform</th>
<th>Select</th>
<th>Dispatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>verify data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>consolidate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>preprocessing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>select method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>process data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>postprocessing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>deliver result</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5. Generalized Process Flow

A lot of external tool support that has the additional advantage of a standardized transport representation (XML).

The second issue for reconfigurability is the binding of input variables to external data sources. We have chosen to use a condition-matching approach, where each input variable has a number of conditions associated with it and if an incoming data element matches them, it will be used as the value of the variable. As our data representation is XML, the XPath[12] navigation language was chosen for this task.

The last aspect for reconfigurable transformations is synchronization of the input data. Data may not always arrive in the same order and sources may have different rates. Therefore, a policy is required that defines when a new data element will be produced from the available inputs. For now, we have chosen the simplest policy: Whenever new input data arrives, and if all variables are filled, a new output element will be produced.

The data fusion itself has been rewritten into an XQuery expression, which provides an extensible, easily scriptable processing language. XQuery could have been used without the FST library, with all the same benefits. However, XQuery requires external API support and some binding code, which would not have been worthwhile to implement if used in just one component. Therefore, it was only undertaken when the prospects of reuse increased.

**B. Communication Mismatch Resolution**

The InterestRater is essentially a pure event-processing component: Its correctness does not depend on the replies of its output components. However, other components might depend on the reply. In particular, the object recognition component is interested in user labels. Therefore, communication with the dialog manager has been transformed into a task-interaction, a special form of request-reply, which allows other components to observe the exchange and monitor whether it is successful or not. In essence, the issue has been off-loaded onto the middleware framework used. However, due to the use of the TDL, this is completely transparent to the core IR functionality.

**V. RELATED WORK**

Although a large number of robotics architectures exist that target re-use as a primary aim, almost all of these differ in terms of the granularity level at which re-use is realized. For instance, the RT-BOS framework [13] aims at a rather fine-grained level of component granularity. In contrast, frameworks like MARIE [14] consider components as rather coarse legacy components that need to be integrated.

A comparable level of granularity appears to be targeted by the Concurrency and Coordination Runtime (CCR), now a part of the Microsoft Robotics studio [15]. The CCR has other similarities to the proposed approach, in that it offers a library of low-level programming elements for managing concurrent processes. However, the CCR does not address
filtering and selection, which we have identified as key elements for reconfigurability.

VI. CONCLUSION

The filter-transform-select (FTS) approach describes a general pattern for arbitrary components that focuses on breaking up components into well-defined sub-parts. Preliminary case studies suggest that the proposed sub-divisions increases the potential for reuse across components.

We have also introduced an initial set of reusable building blocks, both on the fundamental level, where concepts from flow-based programming are picked up and on a domain level, to support typical problems in robotics applications. These building blocks are at a granularity substantially above classes or individual functions but below components. We expect that this granularity is particularly suitable for reuse of parts across architectures and systems. That said, the initial set of components is just a starting point. More research into optimal granularity and other aspects supporting for reuse is required and we hope that our approach will stimulate more discussion, in particular across architectural boundaries.

The representation as a transformation graph provides a basis for external reconfigurability, which reduces coupling and enables external configuration. This will allow components using the FTS library to be adapted to new systems, including those with different architectures, without changes to the code. Additionally, it enables efficient, parallel implementation of the graph execution engine.

A. Future Work

Our next steps will be to consolidate the library and current building blocks and release it as a stand-alone package. An upcoming version of our event-based middleware [16] will also make use of the FTS library for its internal process flow. As part of this effort, an explicit model description is to be developed, which allows easy composition of transformation graphs and exposes configuration variables to the outside.

Furthermore, validation of the concepts by application to more components is a top priority and currently ongoing. This should also result in a bigger library of reusable building blocks that can be included with the library. Similarly, we would also welcome experience reports from application of the library to other frameworks and systems.

Optimization of the library is obviously an important issue if it is to be widely used. While our tests so far show more than adequate performance, moving more application logic into generalized filters and transformations will require further re-evaluation. Fortunately, the transformation graph offers many opportunities for automatic optimization. For example, redundant or partially overlapping patterns can be joined, so that they only have to be matched once. Integration with middleware (any middleware, not just ours) could also lead to some matching processes being “pushed down” to the transport level, to reduce data transmission requirements. Last, but not least, it may also be possible to remove some data transformations when all filters support the original types.

REFERENCES