Requirements and a Case-Study for SLE from Robotics: Event-oriented Incremental Component Construction

Ingo Lütkebohle and Sven Wachsmuth
Applied Informatics Group
Bielefeld University
{iluetkeb,swachsmu}@techfak.uni-bielefeld.de

Abstract: Research in the area of human-robot interaction requires a tight interleaving of incremental system development and experimentation across different robotic platforms. In terms of software engineering this proposes certain challenges to system development that are only partially covered by component-based robotic engineering. For the incremental component composition, we introduce an explicit ganularity level above functions but below components using an event-driven data-flow model. Two different case studies show its impact on the re-use and maintenance of software components. We discuss requirements and possible impact of software languages for the graph-based decomposition approach.

1 Introduction

Robotics is a sub-area of cyber-physical systems that is highly diverse, internally, with hardware ranging from nano-scale robots through robots acting in human environments to industrial and field robots. The software eco-system in robotics is at least as diverse – if not more. Correspondingly, platform independence and re-use across platforms and software frameworks is of high importance in this area. Moreover, research systems in robotics – especially in the area of human-robot interaction – are changing rapidly, in order to explore and test novel ideas. Although, re-use is an enabling key factor, software components practically need significant adaptation before being applied on a new or changed system environment. We believe that language engineering can contribute here, by making re-use more efficient.

Our foundation is the component-based software architecture approach, which is now the dominant mode for building large robot software systems. In the last years, different component models and operating cores have been put forward in this regard, like OROCOS [BsK03], SmartSoft [Sch07], CLARAty [Nes07], and ROS [QCG+09]. These address how to compose a system from components, but they say little about the creation of the components themselves. As in many other areas, components are usually created from both new and existing code. The writing of the necessary composition code is usually done in the same language as the combined code, which is often C/C++ or a similar language. The effort for composing and changing such components is typically underestimated, especially if functionality is not already isolated into a library. In this case, it can be even difficult to extract the functionality for re-use in new components. In order to address these challenges, the development model needs to explicitly support fast changes or re-use. Rapid component construction should make this easier, and in robotics, it can take advantage of the fact that many changes are similar. If a new sensor or a new output is added, many processing steps typically re-occur in different combinations, like data parsing, filtering, fusion, communication setup, or

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4
weitere Artikel online:
http://informatik2011.de/519.html

device access. To exploit this structure for re-use, we developed a graph-based component model, where the nodes have a granularity level above functions/objects but below components. Nodes are implemented in an object-oriented programming language (currently Java), but the exchange of data between them is specified in the graph, which is easily changed. This also makes the input-output interface of the nodes very regular. In effect, the graph makes it easy to add new functionality, and the regular node interface and granularity facilitates re-use.

The main contribution in this paper is to evaluate the actual efficency of this type of component composition in two case studies that are typical in robotics: Firstly, data fusion, which means the combination of sensor data from multiple sources, and secondly, achieving a form of platform independence by implementing hardware control drivers that map differing hardware semantics to a consistent interface. Furthermore, we suggest a novel way to determine node granularity and, finally, introduce our first steps towards an explicit language support for the approach. This last part emphasizes potential improvements through and requirements for a systematic exploration of SLE techniques. Parts 3 and 4 of this contribution are scheduled to appear in [LW11] and are reproduced here for completeness. In both cases, they constitute abbreviated reports on the case studies, with full details available in [Lü11].

2 Software Engineering Aspects of Robotics

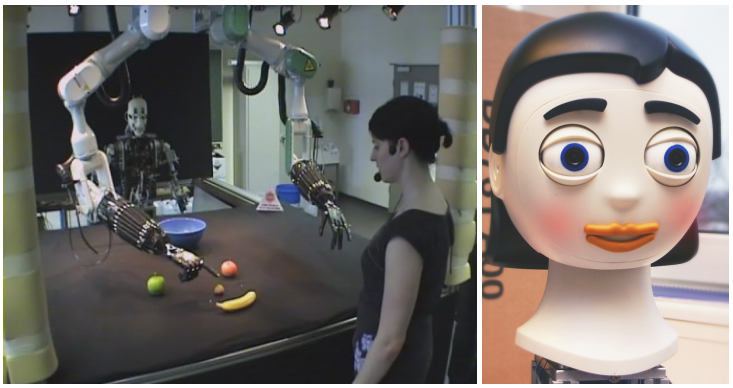
While many aspects discussed so far apply to robotics in general, our own work is in Human-Robot-Interaction and contributes to the goal of enabling more natural interaction using speech and vision. To introduce our assumptions, the next part will shortly sketch the concrete scenario used. We expect that, at least regarding re-use and platform independence, our use-cases are fairly representative for issues faced by other robot software systems. After that, we will give a short overview of related architectural approaches.

2.1 The Curious Robot Scenario

The so-called "Curious Robot" scenario [LPS+09] has the goal to learn object labels and grips in a natural, speech- and vision-based interaction between a human and a robot. Its visible components are depicted in figure 2(a). It is composed out of an anthropomorphic platform (originally the upper-torso-robot BARTHOC [HSF+05]) and a hand-arm system based on the anthropomorphic Shadow hand [RHSR07]. The former of these serves as an intuitive contact point for the human but has no manipulation capabilities, the latter resembles a traditional industrial system, but enables powerful manipulation (in fact, on a level considerably beyond most traditional industrial manipulators). In the basic scenario, the robot identifies interesting object candidates through visual saliency and first asks for their label, trains an object recognizer, asks for the grip type to be applied, and then puts the object away using the specified grip. Even though this is a simple, linear process, both the environment and the speech-based interaction create many diversions, such as non-objects being asked for due to spurious bottom-up detections, speech being mis-understood, gestures being mis-interpreted, the human interrupting the robot during actions, and so on. Dealing with these issues in appropriate manners created many architectural challenges regarding the coordination of the interaction and action sub-systems. Moreover, a parallel sub-system produces social cues, such as gaze feedback, to aid interaction. In more recent work, instead of the BARTHOC torso, we have constructed and applied a new robot head called "Flobi" [LHS+10], depicted in figure 2(b). Flobi sports an expressive face, and has been designed to elicit a sympathetic response, mainly due to its

ISBN 978-3-88579-286-4

Figure 1: The Scenario



(a) The "Curious Robot" asks for a label (b) The "Flobi" head

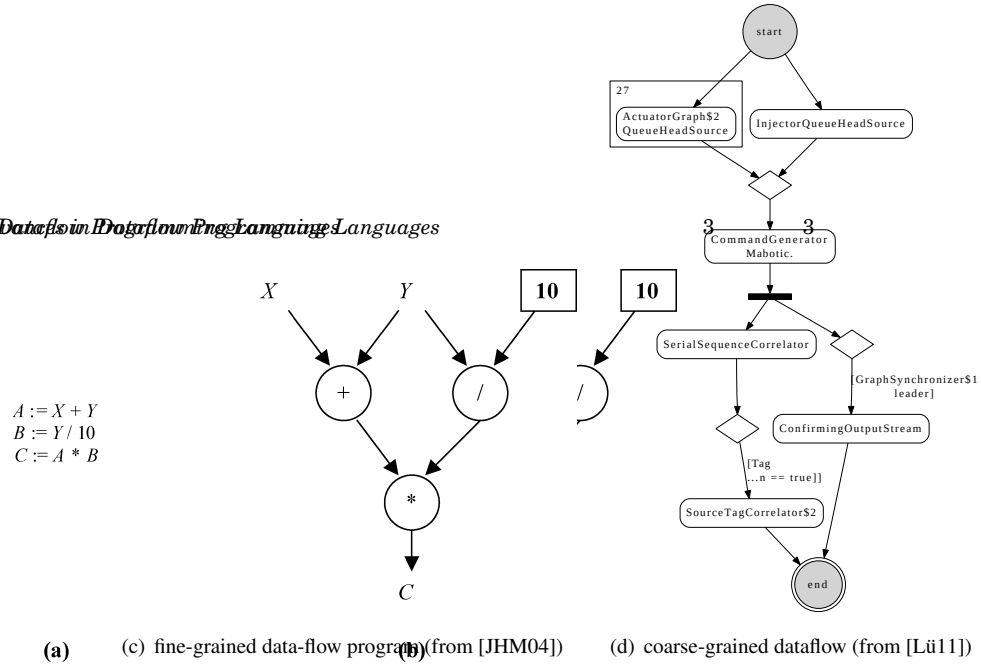
much more friendly exterior [Heg10]. Both the electrical interfaces and the capabilities of Flobi are quite different from BARTHOC, so its application has also created a number of typical challenges for the software architecture, being representative for changing software-hardware interfaces during system evolution.

2.2 Related Architectural Work

The rapid composition of components is a well-known challenge in software architecture. This is particularly noticeable for distributed systems, because of middleware integration, and for research-intensive systems, because of the need to rapidly implement, evaluate and test multiple approaches or iterations. The methods and tools in place for composition directly affect the quality attributes of the resulting software. Not surprisingly, the literature contains a number of proposals to achieve this goal, such as module-interconnection languages (cf. [SG96] for a discussion), data-flow and visual programming (reviewed by [JHM04]), etc. In industry, Model-Driven-Engineering (MDE) is the newest attempt at these ideas. However, as Schmidt points out [Sch06], the non-functional qualities of such tools can be decisive for their applicability. For example, CASE tools were widely seen as too far removed from the domain of application. Every tiny bit had to be modelled, which caused models to become complex and unmaintainable, quickly. MDE, which adds meta-models, is seen as one potential way out, but still unproven. The data-flow approach, on the other hand, has been proven to be practicable in some important domains (such as signal processing, e.g. [LM87] and control, e.g. [BH08]), but it is unclear whether it can also be applied to more general modeling cases and what kind of domain-specific abstractions will be necessary. It is also noticeable that data-flow often comes with powerful tool support, e.g. as provided by TI LabView or Mathworks Simulink.

3 Facilitating Incremental Component Composition

Based on the outlined considerations, we have investigated the data-flow approach for component composition, particularly regarding the software engineering aspects of re-use, maintainability and tool-support. Our approach is based on a toolkit that enables application of the data-flow approach to existing programs, thus substantially



(c) fine-grained data-flow program (from [JHM04])

(d) coarse-grained dataflow (from [Lü11])

Fig. 1. A simple program (a) and its data-flow equivalent (b).

Figure 2: Examples for data-flow programs of differing granularity.

2. THE DATA-FLOW EXECUTION MODEL

2.1. The Pure Data-flow Model

In the data-flow execution model, a program is represented by a directed graph, consisting of nodes and arcs. Nodes represent operations, and arcs represent data-flow paths between operations. Data-flow graphs are also used for compiler optimization and for hardware-level parallelism [JHM04]. Corresponding, the nodes were very fine-grained, down to individual instructions (cf. figure 2(c)). This was identified as a problem, and approaches have emerged that allow larger granularities, with our own approach being one of the latest [Lü11] (figure 2(d)).

The graph in itself does not precisely specify the order of execution. Classically, data-flow is *data-driven*: Whenever the necessary data is available at the input(s) of a node, it will be processed and the result, if any, is placed on the output arc(s). Executable nodes are also called *enabled*, and the data, which is represent as a token, is consumed from the input arcs. It then ceases to be enabled, and waits to become enabled again. Which this model has been newly developed, its implementation is based on widely applied concepts, the difference is only in how we use them. Specifically, we have adapted ideas from event-based integration to suggest the granularity of data-flow nodes, a traditionally difficult issue. We suggest that re-use, in particular, will be increased by our proposal. Based on these ideas, several case studies have been carried out in the area of information fusion, control drivers and such as arithmetic operations. We have examined the level of abstraction, finding substantial improvements in both. Lastly, we have also attempted to identify potential limits of the approach and how to overcome them. One result in this area regards global state management and we will outline a potential mitigation strategy.

Figure 1 shows a simple program (a) and its data-flow equivalent (b). Nodes represent operations and the arcs carrying data between operations. This dates back at least to the 1970s, when it was motivated by hardware-level parallelism [JHM04]. Corresponding, the nodes were very fine-grained, down to individual instructions (cf. figure 2(c)). This was identified as a problem, and approaches have emerged that allow larger granularities, with our own approach being one of the latest [Lü11] (figure 2(d)).

The graph in itself does not precisely specify the order of execution. Classically, data-flow is *data-driven*: Whenever the necessary data is available at the input(s) of a node, it will be processed and the result, if any, is placed on the output arc(s). Executable nodes are also called *enabled*, and the data, which is represent as a token, is consumed from the input arcs. It then ceases to be enabled, and waits to become enabled again. Which this model has been newly developed, its implementation is based on widely applied concepts, the difference is only in how we use them. Specifically, we have adapted ideas from event-based integration to suggest the granularity of data-flow nodes, a traditionally difficult issue. We suggest that re-use, in particular, will be increased by our proposal. Based on these ideas, several case studies have been carried out in the area of information fusion, control drivers and such as arithmetic operations. We have examined the level of abstraction, finding substantial improvements in both. Lastly, we have also attempted to identify potential limits of the approach and how to overcome them. One result in this area regards global state management and we will outline a potential mitigation strategy.

3.1. The Data-flow Model

Figure 1 shows a simple program (a) and its data-flow equivalent (b). Nodes represent operations and the arcs carrying data between operations. This dates back at least to the 1970s, when it was motivated by hardware-level parallelism [JHM04]. Corresponding, the nodes were very fine-grained, down to individual instructions (cf. figure 2(c)). This was identified as a problem, and approaches have emerged that allow larger granularities, with our own approach being one of the latest [Lü11] (figure 2(d)).

The graph in itself does not precisely specify the order of execution. Classically, data-flow is *data-driven*: Whenever the necessary data is available at the input(s) of a node, it will be processed and the result, if any, is placed on the output arc(s). Executable nodes are also called *enabled*, and the data, which is represent as a token, is consumed from the input arcs. It then ceases to be enabled, and waits to become enabled again. Which this model has been newly developed, its implementation is based on widely applied concepts, the difference is only in how we use them. Specifically, we have adapted ideas from event-based integration to suggest the granularity of data-flow nodes, a traditionally difficult issue. We suggest that re-use, in particular, will be increased by our proposal. Based on these ideas, several case studies have been carried out in the area of information fusion, control drivers and such as arithmetic operations. We have examined the level of abstraction, finding substantial improvements in both. Lastly, we have also attempted to identify potential limits of the approach and how to overcome them. One result in this area regards global state management and we will outline a potential mitigation strategy.

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4
weitere Artikel online:
http://informatik2011.de/519.html

tion thus follows the data, as it flows through the graph – hence the name.

The classical model assumes infinite hardware parallelism, which is not realizable. The pure data-driven approach may also waste computation on results which are not needed. Therefore, other models have been suggested, such as Kahn process networks (KPN) [KM77]. These are *demand-driven*, i.e. nodes are only activated when their outputs are requested. Other models, such as SDF [LM87] use node-metadata to statically schedule execution as needed. However, these approaches also come with complexity, are not always realizable (e.g. SDF still uses dynamic scheduling for conditionals) and are primarily necessary for fine-grained data-flow. Therefore, in our approach, we have opted for the classical data-driven execution. Our toolkit supports both one-thread-per-graph (optimized for minimal scheduling overhead) as well as one-thread-per-node (optimized for minimal latency) execution models.

3.2 The Filter-Transform-Select Decomposition Principle

The granularity of nodes has a major effect not only on the execution but also on modeling and understanding of dataflow graphs. Our primary concern is the reduction of unnecessary complexity, while increasing functional re-use. Taking inspiration from event-based integration [BCTW96], we suggest a decomposition into *filters*, *transformations* and *selection* (FTS) nodes [LSW09]. These are defined as follows:

Filter. A filter decides whether a particular graph section is used. This is directly relevant for re-use, as we may want to re-use functionality in different circumstances than originally envisioned.

Transform. The processing nodes themselves are transformations.

Select. Whenever there are multiple options, at least one of which must be taken, a select node chooses the path. It is an error if no condition is met, which promotes a *structured dataflow* design that reduces mistakes.

Besides these distinctions, the decomposition also depends on the level of interest. For example, an algorithm often contains many conditions. As outlined above, describing a graph structure on that level is considered too fine. Therefore, the decomposition into filters, transformations and selectors is suggested to be performed at the *highest possible level*, just below the component boundary.

As a last aspect, we have also carried forward these distinctions into a domain-specific-language for the creation of graphs. It separates the implementation from the model and, due to its higher level of specification, supports rapid application development. While our language is just a prototype so far, it has already been used successfully for several of the case studies presented in the following.

4 Case Studies

Our main interest has been to evaluate the applicability of the data-flow approach across several domains. Therefore, we have conducted several case studies, where we compare conventional implementations with FTS-based ones, as well as successive iterations of data-flow applications. These allow us a quantitative assessment of the level of re-use seen and also feedback on the maintenance operations necessary, i.e. the maintainability.

It should be noted, however, that the intent of these comparisons is *not* to make absolute value judgments – far too many variables would be a factor. Instead, the intent is to gain insight into what kind of issues to expect when *changing* between approaches. In the following, we will summarize the case studies and present our

ISBN 978-3-88579-286-4

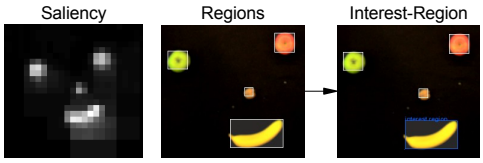


Figure 3: Typical input data (from [LPS+09]).

conclusions. The full studies are available as part of a PhD thesis [Lü11]. The graphical representation of models is based on UML activity diagrams [UML05], with plate model extension [Lü11].

4.1 Data fusion

The first object of study has been the action selection component in the "Curious Robot" scenario [LPS+09, LPS+11]. Action selection is an important part of any autonomous system. In such components, communication and interaction with other components, as well as data fusion often make up a substantial part of the code. These are exactly the areas where we expect re-use potential. Furthermore, data fusion is typically tree structured, which constitutes a baseline processing case.

To realize action selection in this instance the following functions are necessary: Visual event receiption, ranking of visual regions, integration of background knowledge, selecting the most salient region and proposing an action that acquires the next most interesting piece of information. See figure 3 for example inputs and results.

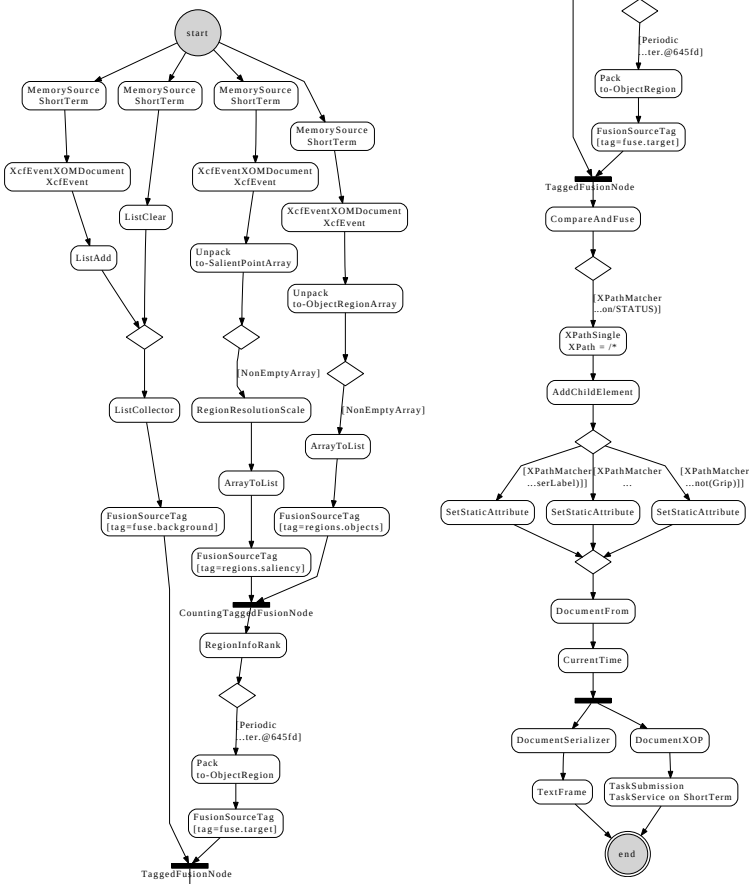
An initial, independent version of the action selection component has been implemented by a colleague of the authors, for the first iteration of the "Curious Robot" demonstrator [LPS+09, section 2.D]. The implementation is small-to-mid-size, with 778 source lines of code distributed amongst 22 classes, 5 of which are small inner classes. From the code size, the basic COCOMO [Boe84] model would put such a component at 1.87 person-months, without overhead. While this may be an overestimate, given that it assumes a typical commercial development process, the component interfaces with many others, adding developer communication, so it seems realistic.

Re-use The component uses a middleware and associated marshaling routines. Otherwise, its implementation is primarily influenced by communication and data-management. The action selection implementation itself is simple, and thus fairly small, and distributed amongst the other code. This is not an ideal situation, if the selection algorithm would need to be upgraded.

After re-implementation using a data-flow formulation, the resulting graph consists of 39 nodes, with 25 different types, containing 696 lines of code. Most importantly, only two node types are specific to the component, the other 23 are potentially generic and 13 (52%) of them have already been re-used in other, unrelated applications. The remaining two nodes contain only 103 lines of code, with another 131 lines in the graph model (using verbose XML syntax).

This high-level of re-use is consistent with expectations about such components, but has only been made actual through the data-flow based composition method. The resulting graph is shown in figure 4.

Maintenance Having demonstrated a substantially increased level of re-use, we now consider maintenance aspects. For this, we have added visualization functionality, displaying the inputs and choices of the component. This is mainly needed for debugging, so we have realized it as an optional, second application. Ordinarily, such



(a) upper part (b) lower part

Figure 4: Visual representation of the action selection component model.

a choice would result in substantial reimplementation, processing the same inputs, etc. However, in the data-flow approach using a textual model, all nodes can be re-used and even the model only needs few modifications.

Specifically, to add visualization, 5 new node types have been added, mainly for display and coloring. The modification operations consisted of three node insertions, an addition of a graph branch (for bringing up the GUI) and a removal of another branch (which used to send out results). We consider this to be a very small number of operations, thus the approach supports maintenance very well.

4.1.1 Case study 2: Hardware independent serial robot control

After looking at data-fusion, which is a popular but not particularly robotics-specific function, the second case-study will consider a subject closer to robotics, motor control through a serial link. Such links are popular, as they allow to implement control

algorithms in a dedicated, real-time capable microcontroller with fairly low effort, but still control them in detail from a more powerful host PC. It requires implementing a control protocol, however, which can be specific to the controller manufacturer.

The requirements for such protocols are as follows:

- Transduce abstract control commands to the vendor protocol and inversely for sensor data. This may sometimes be dependent on current device state.
- Manage access to the serial link – many protocols prohibit sending a command before a reply for the previous one has been received.
- Realize a blend mode – protocols differ in whether a new command overwrites a previous one, is queued or is refused. Some protocols allow a choice, in others this must be realized through explicit queuing or cancellation.
- Deliver feedback information on start of command execution.

Study Design For this case study, a hardware independent layer for different robots has been implemented. The robots are:

1. The BARTHOC humanoid torso [HSF+05], manufactured by MABOTIC GmbH.
2. The Sony EVI D31 Pan-Tilt-Zoom camera, using the VISCA protocol [Son99]. This camera is common in many robotic applications and the protocol is also used for several other cameras by Sony.
3. The "Flobi" anthropomorphic robot head [LHS+10], developed jointly by Bielefeld University and MABOTIC GmbH. It uses an advanced protocol, based on ideas from both of the above.

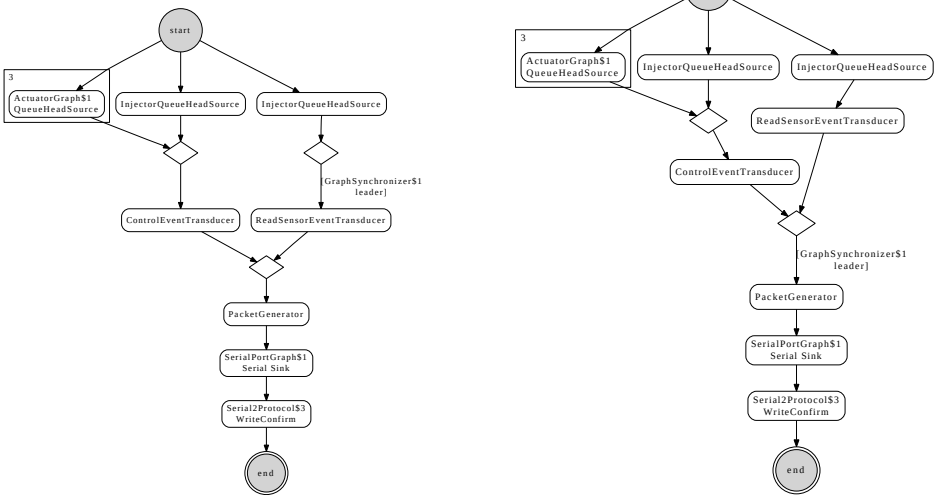
All of the above have been realized using the proposed approach. Regarding reuse, we have examined the size and make-up of the resulting graphs, summarized in table 1. More details and the full graphs are available in [Lü11].

Protocol	#Nodes	#Node types	#Links	#Custom nodes
MABOTIC	11	11	10	2
VISCA	31	25	51	14
Flobi	14	12	20	5

Table 1: Protocol graph size summary

Reuse From this summary, it is obvious that the MABOTIC and Flobi protocols could be realized with moderate effort and using only very few custom nodes. The custom ones are essentially just the transducers for input and output. The rest of the nodes could be re-used between protocols or from the standard node libraries. Such standard nodes include serial i/o, data combination, type filters, etc.

In contrast, the VISCA protocol required a much higher number of node types, much less of which are from the standard set. Furthermore, these nodes are also much more connected, in contrast to the earlier two protocols, which are mostly chains. The reason behind this is that the VISCA protocol keeps considerable state on the controller, some of which is necessary in creating command packets. This unfortunate fact requires the protocol implementation to keep a global state, which leads to a fair amount of connections in order to shuffle this state around. It shows that data-flow modeling, with its local view of data, is not well suited to a global state.



(a) v1: read sync only

(b) v2: all commands synced

Figure 5: Command output graphs for Flobi protocol revisions.

Maintenance One other interesting aspect of modeling stems from a look at maintenance in the Flobi protocol. This model underwent a change during development where, at first, only one type of command caused a reply packet. This created problems when packets were lost due to noise and, consequently, all packets received a reply. In the graph-based implementation, the necessary synchronization changes could be accommodated in the write graph purely by re-wiring it, as shown in figure 5. Please note the edge labeled "GraphSynchronizer" – it is responsible for link access management and, in the second version, has simply been moved so that it intersects both paths.

4.2 Summary

The approach has been experimented on a data fusion with visualization application and on three different robot control protocols, across three different robot platforms. Both the re-use and the flexibility goals could be demonstrated in these case studies. As expected, the benefit of flexible graph construction was particularly visible during evolutionary development, underlining the suitability for rapid development.

We also saw that global state management is not a strong suit of the approach, whereas local state is no issue and regularly used. This suggests that it should be coupled with a coordination approach that manages global state, and couple it to the data-flow graph engine, e.g. to select paths based on external information.

It should also be noted that the case studies presented here only represent a part of the applications realized through such means. Since the toolkit's inception, we have explored a variety of applications. Here, the use of the XML-based textual modeling format has proven particularly useful to rapidly create and modify components.

XML Element	Attributes	Description
`node`	type, *name*, *source*	Node of the given type, with optional name and source node. If 'source' is not given, uses predecessor in document order.
`arg`	type	Configuration argument for construction.
`select`	-	Enclose a number of alternative paths.
`target`	-	One of a number of alternatives.
`filter`	type	Condition to be met for this path.
`fuse`	sources, *required*, *outputType*	Combine output from *n* named nodes into a map or pair.

Table 2: Slightly abbreviated summary of XML elements for model specifications.

5 Constructing Graphs

Originally, the component graphs have been created using the toolkit implementation language (Java). This approach is still used at some places, but several drawbacks were reported by users of the toolkit. Most importantly, constructing the graph and specifying parameters for the nodes is a *configuration* task which is conceptually quite different from the implementation of nodes. Furthermore, from a program understanding point of view, it might be too easy to bypass the intended method of passing data (edges) though shared state variables. These make it much harder for the engine to ensure scheduling constraints, and they also add complexity to analysis and monitoring.

5.1 XML syntax for data-flow graphs

As a first step to address these issues, an XML-based configuration language has been created, based on four constructs: i) node configuration (including naming), ii) implicit and explicit linkage between nodes to create edges, iii) fusion constructs to combine data, and iv) selection. See table 2 for an overview, listing 1 for a simple example and figure 6 for the resulting graph structure.

Listing 1: Example graph processing two different XML-based sources

```
<model>
  <node name="fib" type="XMLFibonacciSource"/>
  <node name="time" source="null" type="XMLTimestampSource"/>
  <select name="choice1" source="fib,time" selectMax="1">
    <target>
      <filter type="XPathFilter">
        <arg type="StringChild">/fibonacci</arg>
      </filter>
      <node type="XPathTransformSingle">
        <arg type="StringChild">number(/fibonacci/value)</arg>
      </node>
    </target>
    <target>
      <filter type="XPathFilter">
        <arg type="StringChild">/*</arg>
      </filter>
      <node type="XPathTransformSingle">
        <arg type="StringChild">string(.)</arg>
      </node>
    </target>
  </select>
  <node name="static" source="null" type="StaticSource">
    <arg type="StringChild">something else</arg>
  </node>
  <fuse sources="choice1,static" required="choice1,static"/>
  <node type="QueueSink"/>
</model>
```


ISBN 978-3-88579-286-4

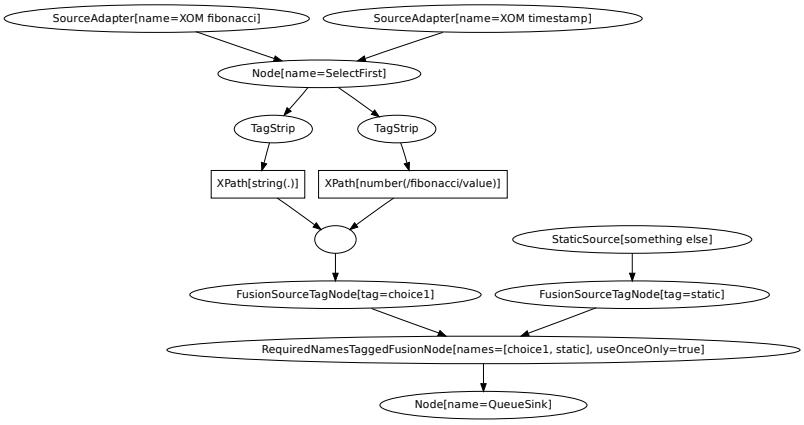


Figure 6: Graph corresponding to listing 1

At this moment, our graph specifications usually run to between 100 and 700 lines of XML, with much of that taken up by configuration expressions. These represent components that, while sometimes not overly complex, are still fully running applications. For example, the robot behaviors from [Lü11, chapter 8 and 9], are realized using such models. The present author has so far used about 140 different node types, with functionality such as middleware connectivity (three different ones), data marshaling, XML parsing, selection and transformation, simple GUIs, finite-state-machines, file and serial line I/O, as well as some utility types, for data manipulation, fusion, and selection.

6 Discussion

While the proposed approach is certainly not a widely used project so far, it has now been successfully applied in several projects at Bielefeld University for about three years, by both students and researchers. In these projects, it has been easily adopted, largely due to its low cost of entry, and use of familiar concepts. Based on this experience, we will now discuss some aspects that we consider of particular interest to Software Language Engineering.

6.1 Granularity

We have initially focused the framework on the granularity issue, by suggesting the FTS decomposition, because we believe that addressing it is key to achieve re-usability: Re-usable chunks should be large enough to be worth-while, but not so large as to be overly specific. That said, we do not claim to have identified the one true granularity. Granularity is always a matter of the detail level: What is a pure transformation on one level of view may very well have many conditions and branches contained in its implementation. Thus, the FTS decomposition is more about extracting the filter that determines when to apply a transformation at all. This is the part that we have seen to change across different domains.

Otherwise, when looking at multiple similar components, determining good granularity may not be as difficult as initially thought: Pick blocks that re-occur across applications. Here, the main contribution may very well have been to supply a framework that emphasizes this issue and, thus, both makes people aware that composing components as a graph of small parts may be a good approach, and assists them in doing so.

6.2 Re-use and Configuration

As the case studies presented have shown, the proposed approach could achieve its re-use goal. It has been asked whether this is due to a second system effect: A later re-implementation could benefit from an improved understanding of a good problem decomposition. This may be, but for several aspects that have seen increased re-use, such as middleware communication, even the developers of the original components already had a good understanding of the problem.

We believe that the seperation of configuration, composition, and computation is more crucial. The seperation of composition from computation has been the initial point of adopting a data-flow approach and, thus, has been expected. The more interesting effect is based on the use of configuration expressions. These have been an almost inadvertant side-effect of the use of a textual specification language: Because arguments could not be constructed from multiple objects as easily, a textual expression language has been added instead. In some cases, such a language has already been used before, e.g. we widely used XPath for conditions and selectors on XML documents. This experience and its good results, then, led to the adoption of the JEXL expression language for specifying conditions on other kinds of objects.

6.3 Experiences with the graph specification language

While the described XML syntax has only been intended as a quick stepping stone, some of its design choices have shown themselves to be both unexpectedly convenient and unexpectedly inconvenient.

Implicit connectivity In contrast to existing graph specification languages, our specialized language has been designed for using implicit linkage between nodes as much as possible. In this mode, unless specified otherwise, two node definitions that follow each other in document order are linked into a chain. When we set out to design the specification, our graphs primarily contained chains, with junction nodes being much rarer. Therefore, using the XML document order when no explicit predecessor is given saves typing, and also makes re-ordering and inserting nodes easier.

In fact, after having used this implicit way of making connections, we consider it indespensible, and would ask for it in any future tool. A comparison to graphical tools has been illuminating here. In principle, one would expect a visual tool for creating graphs to be very natural, and we believe that a visual tool would hold much promise for our approach, too. However, many graphical tools for creating graphs use a boxes-and-arrows model, where edges have to be explicitly specified, and changed, whenever connectivity changes. This makes, for example, moving a node a six-step operation (1: remove incoming link, 2: remove outgoing link, 3: link previous neighbors, 4: remove outgoing link from new predecessor, 5: link new predecessor as incoming, 6: link node to successor), whereas with implicit linkage, it is a two-step operation (1: cut, 2: paste). The same holds for other manipulations. In our opinion, visual tools thus need specialized means to achieve connectivity.

ISBN 978-3-88579-286-4

Verbosity and Generation Using XML as the syntax creates a fairly verbose specification syntax which is comparatively tedious to create. While the approach has served us reasonably well so far, we expect that scaling it up to larger components will likely require different methods.

That said, in several important cases, our current graph specifications substantially consist out of configuration expressions, not XML. For example, we embedded state-machine specifications, or XQuery expressions (XQuery, while being designed for XML processing, is essentially on an SQL-level of verbosity). For such cases, we consider the XML overhead to be insignificant.

On option to shorten the specifications, while keeping XML, that have already experimented with is generating the specification from a more domain-specific specification language, which is expected to result in shorter specifications. The tool support for XML, such as mature support for XSLT transformations, has made such generation approaches straightforward.

Wiring for data combination While the two points above have been unexpectedly positive, one aspect of the current syntax has also proven an unexpected issue: Collection manipulation for data combination. The way we have implemented this is that whenever a node requires more than one input element, these will be fused into a Map with string keys by the "fuse" element. One unfortunate consequence of this implementation choice is that it requires more knowledge about the implementation of a node at configuration time (the expected key names), which is also harder to determine automatically, in contrast to the type information for construction, which is available through reflection. Moreover, once fused, a map may require taking apart again, for nodes that process only one of its elements. At the moment, the specification language has no explicit support for such operations, requiring them to be carried out through transformation nodes. This is clearly sub-optimal.

At the moment, we are undecided as to how to address this best. While it may appear a simple solution to implement nodes with multiple arguments, we have some reservations on the grounds of clarity and predictability. In general, however, language support to move this collection manipulation into the background is definitely desirable and planned as a next step.

6.4 Language Tools

One noticeable draw-back when using XML for specification is a lack of modern development tool support. For example, there is no completion or checking for type names (which represent the operations, and are essentially an open set, hence not easily specifiable using schemata), and no suggestion of argument values. While this is not very different to how much programming used to occur before the advent of IDE's, nowadays it feels decidedly inconvenient. We expect that such issues are rather easy to solve, but they add to the development burden for a DSL.

A more serious issue is that the data-flow approach changes debugging. All user-code is called by an external engine, similar to the Inversion-of-Control style of user interface toolkits. Tracing code execution through this engine is very different from traditional debugging, because user code and framework code is interleaved tightly. This is somewhat offset by the fact that the separation of functionality into well-defined building blocks can make other development tasks, such as unit-testing, easier. However, it is still by no means optimal. A debugger with an awareness of the different execution model imposed by the data-flow engine would be one approach to addressing this issue.

ISBN 978-3-88579-286-4

7 Conclusion

We have presented a graph-based approach for composing components from re-usable building blocks, that achieves higher re-usability by introducing a middle granularity level, below full components, but above the functional level. To complement this, we have proposed a decomposition approach based on event-based systems, the filter-transform-select (FTS) decomposition. In several case studies, we could show that this approach can lead to drastically improved re-use.

Furthermore, we have presented experience reports on using a simple, XML-based language to specify the component graphs. This language aids construction, by providing an easily changed configuration language. Furthermore, it is based on the concept of implicit linkage, which we believe to be an essential feature for data-flow configuration languages, both textual and graphical. From these reports, we have provided some requirements for future work in software language engineering that will be valueable in this direction.

References

[BCTW96] Daniel J. Barrett, Lori A. Clarke, Peri L. Tarr, and Alexander E. Wise. A framework for event-based software integration. *ACM Trans. Softw. Eng. Methodol.*, 5(4):378–421, 1996.

[BH08] B. Bauml and G. Hirzinger. When hard realtime matters: Software for complex mechatronic systems. *Robotics and Autonomous Systems*, 56(1):5–13, January 2008.

[Boe84] Barry W. Boehm. Software Engineering Economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, January 1984.

[BsK03] H. Bruyninckx, P. soetens, and B. Koninchx. The real-time motion control core of the orocos project. In *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA)*, pages 2766–2771, Taipei, Taiwan, Sept. 2003.

[Heg10] Frank Hegel. *Gestalterisch konstruktiver Entwurf eines sozialen Roboters*. PhD thesis, Bielefeld University, 2010.

[HSF+05] M. Hackel, St. Schwope, J. Fritsch, B. Wrede, and G. Sagerer. A Humanoid Robot Platform Suitable for Studying Embodied Interaction. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*. IEEE, 2005.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[KM77] Gilles Kahn and David MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing '77: Proceedings of IFIP Congress*, pages 993–998, Amsterdam, The Netherlands, 1977. North-Holland Publishing Co.

[Lü11] Ingo Lütkebohle. *Coordination and Composition Patterns in the "Curious Robot" Scenario*. PhD thesis, Bielefeld University, 2011. in press.

[LHS+10] Ingo Lütkebohle, Frank Hegel, Simon Schulz, Matthias Hackel, Britta Wrede, Sven Wachsmuth, and Gerhard Sagerer. The Bielefeld Anthropomorphic Robot Head "Flobi". In *2010 IEEE International Conference on Robotics and Automation*, Anchorage, Alaska, 2010. IEEE.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9), 1987.

[LPS+09] Ingo Lütkebohle, Julia Peltason, Lars Schillingmann, Christof Elbrechter, Britta Wrede, Sven Wachsmuth, and Robert Haschke. The Curious Robot - Structuring Interactive Robot Learning. In *International Conference on Robotics and Automation*, Kobe, Japan, May 2009. Robotics and Automation Society, IEEE.

[LPS+11] Ingo Lütkebohle, Julia Peltason, Lars Schillingmann, Christof Elbrechter, Sven Wachsmuth, Britta Wrede, and Robert Haschke. Realizing a Robot System for Interactive Online Learning. In *Towards Service Robots for Everyday Environments*. Springer Verlag, 2011. forthcoming.

[LSW09] Ingo Lütkebohle, Jan Schaefer, and Sebastian Wrede. Facilitating Re-Use by Design: A Filtering, Transformation, and Selection Architecture for Robotic Software Systems. In *Workshop on Software Development and Integration in Robotics*, Kobe, Japan, 2009.

[LW11] Ingo Lütkebohle and Sven Wachsmuth. Event-oriented Incremental Component Construction. In *Towards Service Robots for Everyday Environments*. Springer Verlag, 2011. forthcoming.

[Nes07] I.A. Nesnas. The clarity project: Coping with hardware and software heterogenity. *Software engineering for experimental robotics (Series STAR)*, 30, 2007.

[QCG+09] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[RHSR07] Frank Röthling, R. Haschke, Jochen J. Steil, and Helge J. Ritter. Platform Portable Anthropomorphic Grasping with the Bielefeld 20-DOF Shadow and 9-DOF TUM Hand. In *Proc. Int. Conf. on Intelligent Robots and Systems (IROS)*. IEEE, 2007.

[Sch06] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *COMPUTER*, 39(02):25–31, 2006.

[Sch07] C. Schlegel. Communication patterns as a key towards component interoperability. *Software engineering for experimental robotics (Series STAR)*, 30:183–210, 2007.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996.

[Son99] Sony Corporation, 4-16-1, Okata, Atsugi-shi, Kanagawa-ken, 243-0021 Japan. *Command list – Intelligent Communication Color Video Camera EVI-D30/D31*, v1.21, english edition, 1999.

[UML05] Unified Modeling Language: Superstructure version 2.0. Technical report, Object Management Group (OMG), Inc, 2005.