Controller Patterns for Component-based Reactive Control Software Systems

Petr Štěpán, Kung-Kiu Lau
School of Computer Science, The University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
pstepan,kung-kiu@cs.man.ac.uk

ABSTRACT

It is considered good practice in control software design to distinguish computation and coordination on the architectural level. Current component models largely fail to provide distinct abstractions for that purpose. In this paper, we introduce such distinct abstractions. In particular, we introduce controller patterns, an abstraction for defining coordination in the context of component-based software development. We present their definition and demonstrate their usage in a case study, conducted in our prototype tool.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Process control systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures

Keywords

reactive control software, computation and coordination separation, coordination patterns, composite connectors

1. INTRODUCTION

Reactive systems [6] are systems that continuously react to their environment. Their behaviour can be conceptualised as an infinite cycle of reading inputs from the environment, computing the reaction of the system, and outputting the reaction back to the environment. A particular subclass of reactive systems, prevalent in the domain of embedded systems, are systems managing the operation of the device they are embedded in, hence we call them *reactive control systems*. The general schema of a reactive control system is depicted in Figure 1. The figure shows a system embedded in an environment that it controls via actuators and monitors through sensors. Examples of reactive control systems vary from cruise control systems in cars to control systems preventing core meltdown in nuclear power plants.

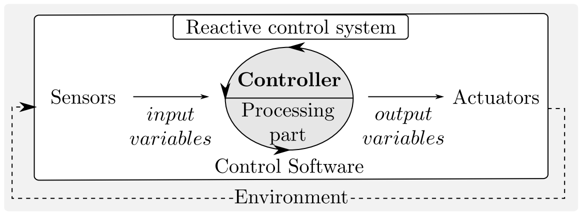


Figure 1: Reactive Control System

The subsystem defining the system's reaction to data coming from sensors is often implemented in software. In this paper, we are interested in the component-based development of such control software.

In general, control software comprises a mixture of computation (data transformation, expression evaluation) and coordination (determining the flow of control and data). It is well acknowledged in this domain that it is beneficial to separate the two on the level of architecture, by defining them as different architectural entities. For instance, Selic [15] calls the principle 'separation of control from function' and defines an architectural pattern named Recursive Control Pattern. The core idea of the pattern, shown in Figure 2, is

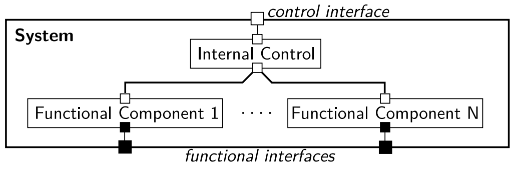


Figure 2: Recursive Control Pattern [15]

to have some components responsible for coordination and other components responsible for computation. We denote the former as a *controller* and the latter as the *processing part* of a control software system (see Figure 1). Similar ideas can be found throughout the domain. Labbani et al. show such separation in Scade [8], Lea distinguishes between functional and controlling components in avionics control systems [13], Haslum identifies 'mode switchers' and 'higher-order task procedures' patterns for reactive programs [7], etc.

In component-based software development (CBSD), there exist two basic architectural abstractions for constructing systems: components and connectors. Conceptually, components are units of computation; whilst connectors deal with interactions between components [17]. Admittedly, connectors have received less attention than components and are not even first-class entities in many component models [12, 2]. As a result, in current component models used in the domain of reactive control systems, controllers end up being defined by components, semantically indistinguishable from processing components.

We believe that it is beneficial to use different abstractions for defining controllers and processing components. For controllers, we propose to use connectors that encapsulate control flow and data flow to coordinate processing components (i.e. use connectors to perform coordination service [14]). Moreover, such connectors, which we call *controller patterns*, need to be composable in order to be able to express potentially complex coordination required of a controller.

Controller patterns allow for distinguishing between controllers and processing components as semantically different entities on the level of architecture. Moreover, due to their coordination nature, the coupling between controller patterns and their coordinatees is looser than that of corresponding component representations of controllers. This leads to an increased reuse potential. Unlike architectural patterns, e.g. the Reactive Control pattern, controller patterns, as component model elements with precisely defined semantics, can be used directly in a CBSD way of constructing control software.

In this paper, we define controller patterns as connectors in the context of a control-driven and data-driven component model that we defined previously [11]. In Section 2, we show the inadequacies of components in current component models used in the considered domain for representing controllers. After giving an overview of our approach in Section 3, we define controller patterns[1] in Section 4. Finally, we apply our prototype tool to an example of a climate control system in Section 5.

2. DESIGNING CONTROLLERS IN CURRENT COMPONENT MODELS

In this section, we consider how current component models support the design of controllers. We show that most of component models in the domain cannot represent controllers using connectors and we show why components are not the most appropriate abstraction for doing so. We illustrate our points on a generic architecture description language (ADL) [12], as a representative of a large group of component models based on port connection, and two component models used in the domain of embedded systems: a research component model for control-intensive systems, ProCom [16], and an industrial tool for designing reactive systems, Scade [3].

2.1 Controllers as Connectors

In order for a component model to represent controllers as connectors, the model has to feature a set of connectors expressive enough to define patterns of control flow and data flow that constitute the controller's functionality. Furthermore, the connectors need to be composable to accommodate possibly complex coordination strategies that controllers encapsulate.

[1]For lack of space, the definition of controller patterns in this paper does not include the specification of their execution semantics, which extends the original execution semantics of our component model, presented in [11].

However, connectors in most of current component models are far from fulfilling these requirements, and thus cannot be used to define controllers. Mehta et al. [14] list in their connector taxonomy the types of connectors dealing with communication and coordination: procedure calls, events, data access and streams. This understanding of connectors as thin wrappers over the underlying middleware communication mechanisms is wide-spread among component models. The connectors there do not encapsulate any complex control flow and data flow patterns and cannot be composed to create such patterns.

A generic ADL has implicit connectors abstracting away procedure calls, which cannot be composed. ProCom is also based on port connection, but distinguishes between control and data connections. Additionally, it features more advanced connectors for routing data or control outside of components. However, the set of basic connectors is not Turing complete (sequencing and looping are not explicitly expressed as connectors but only implicitly by control links) and thus is not expressive enough. Moreover, although connectors can be connected together in an architecture, they cannot be dealt with as a single abstraction. They can only be contained within a composite component. Finally, Scade only features non-composable data flow connections between components. Control flow cannot be expressed in Scade explicitly.

Component models that use coordination as their composition mechanism are more suitable for expressing controllers. Lau and Rana [10] identify web services and X-MAN component model [9] as members of this category. The former is focused on the interoperability of services on the web and thus does not match the needs of reactive control systems. The latter contains a Turing complete set of composable connectors that define control flow. Although X-MAN connectors are suitable for our purpose, data flow representation is implicit in X-MAN, which limits its expressiveness.

2.2 Controllers as Components

Component models that cannot represent controllers as connectors have to define them as components. Such representation of controllers has several drawbacks:

Firstly, controllers are indistinguishable from processing components in a system architecture despite the fact that they are semantically different. This diminishes the understandability of the architecture and is also prone to breaching the desirable separation between coordination and computation, as depicted in Figure 2.

Secondly, controller components may have unnecessary dependencies, not mandated by the controller logic but rather by limitations imposed by the composition mechanism of a particular component model. For instance, in a generic ADL a controller component must have a dependency on the components it coordinates, via required ports. It could not be reused to impose the same control flow and data flow coordination pattern on components exposing interfaces other than those specified by its required ports.

Thirdly, the means for representing the controller's logic may not be sufficiently expressive. This is the problem of Scade. Since all connections represent data flow, explicit control flow between components cannot be expressed. Thus, Scade's expressiveness in representing controllers is limited to data flow routing only.

3. OVERVIEW OF OUR APPROACH

Motivated by many architectural patterns calling for the separation of controllers and processing components and the failure of current component models to fully achieve it, we aim to define a new abstraction for defining controllers: controller patterns. The idea is that each controller pattern represents some coordination behaviour that manages a group of processing components in terms of the order of their execution and data exchange. That is, controller patterns encapsulate control flow and data flow to coordinate a group of processing components. The processing components are not part of the pattern, they are only parameters of the pattern template, analogous to pattern participants in design patterns [4]. As templates, controller patterns are meant to be useful and reusable in different contexts, coordinating different components. Another important characteristic of controller patterns is that they can be put together to form more complex coordination behaviour. Ultimately, their composition defines a controller.

To realise controller patterns in the CBSD context, we need a component model that (i) allows for control flow and data flow to be explicit in the architecture and separate from computation, and (ii) provides the means for the composition of the flows. We have defined a component model satisfying the former requirement in [11]. The following subsection gives a summary of the model. Next, we illustrate the main ideas of the approach on a simple controller example.

3.1 Our Component Model

The component model strictly separates computation, encapsulated in components, from the flow of control and data between components, which is represented by connectors. Further, the model contains data coordinators, entities responsible for dynamic routing of data flow within a group of data connectors.

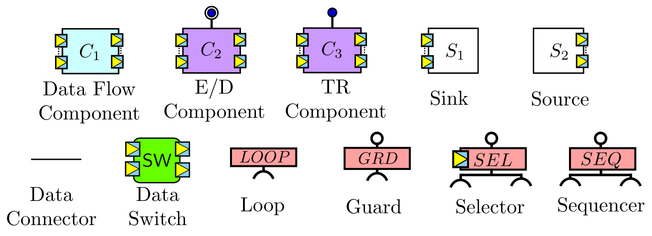


Figure 3: Component Model Elements

A summary of the component model elements is given in Figure 3. Components in our model perform a single function, transforming input values coming from their input data ports to output values that are written to their output data ports at the end of the component execution. Components do not call each other. Data flow components have only data ports and they are data-driven (e.g. SensorProcessor in Figure 4). E/D components are data-driven components but they can be disabled by control through their control port. TR components (such as Motor up in our example), on the other hand, are only executed when their control port is triggered by control signals. The data exchange with sensors and actuators is done through source and sink components, respectively.

The job of transferring data and control in a system, and thus coordinating computation performed by components, is delegated to connectors. There are two kinds of connectors distinguished by the type of flow they carry: data and control connectors. Data connectors are unidirectional channels connecting an output data port to an input data port. They vary in their capacity and the semantics of writing and reading values to/from them.

Control connectors coordinate control flow in a system. Their interface is formed by a control port, one or more control parameters and, in some cases, by input data ports that provide connectors with the data needed for control coordination. As shown in Figure 3, there is a fixed set of control connectors, corresponding to the basic control structures of sequencing (sequencer), branching (selector and guard) and looping (loop).

The last category of elements of this model are data coordinators. Their interface comprises data ports only. They route data flow within a group of connected data connectors. Each of the coordinators realises a particular routing behaviour.

3.2 An Illustrative Example

In this paper, we introduce composite connectors to our component model, i.e. connectors composed of other, simpler connectors. By doing so, we gain the ability to define complex controller patterns. Figure 4 illustrates a real controller pattern used in a window controller system. The system prevents the window motor from being activated when the window is already in the requested position and, additionally, assumes the button can control motor movements in both directions.

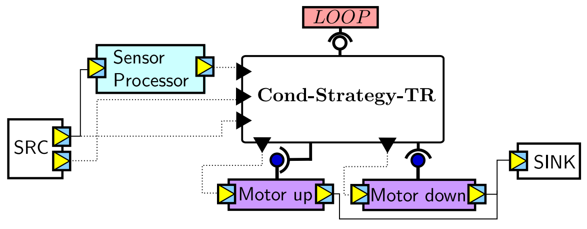


Figure 4: The Cond-Strategy-TR Controller Pattern in the Window Controller System

The architecture shows a clear separation between the processing components (sensor processing and computing commands for the motor performed by SensorProcessor, Motor up and Motor down, respectively) and the controller consisting of the Cond-Strategy-TR controller pattern and a loop. It is easy to see that the system architecture complies with the Recursive Control architectural pattern in Figure 2.

Cond-Strategy-TR is a composite connector in our model and defines the controller's behaviour. It has an interface for data entering and leaving the connector (the filled triangles) and similarly for control (the lollipop and receptacles). It uses data from SensorProcessor to decide whether the motor needs to be activated and if so, it selects the direction of the motor's movement by routing the control coming from the Loop to a respective TR component. The pattern functionally resembles the Strategy pattern [4] as it chooses the strategy which defines the system's behaviour. Additionally, the strategy selection is conditional. Hence, its name – Cond-Strategy-TR.

4. CONTROLLER PATTERNS

In our component model, we can define controller patterns, and ultimately controllers, as composite connectors. Like controller patterns, composite connectors can define complex coordination patterns since they are hierarchical compositions of connectors and data coordinators. In this section, we first present their interface and then discuss how they are constructed.

4.1 Interface

Every composite connector has an interface that determines how it interacts with other entities in a system architecture. The interface is comprised of required data ports, a control port, and control parameters (see Figure 5). The control port is the locus for the control coming from a superior coordinator; while control parameters are connected to subordinate coordinatees. The required data ports define entry and exit points for the data routed by the connector. Each required data port specifies a constraint, in terms of type and directionality, that has to be satisfied by a plain data port connected to the required data port during instantiation of the connector.

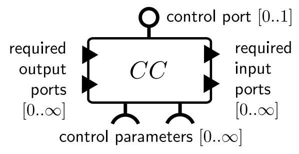

control port [0..1]
required output ports [0..∞]
CC
required input ports [0..∞]
control parameters [0..∞]

Figure 5: Composite Connector Interface

Figure 4 shows an instantiated Cond-Strategy-TR connector: control ports are connected to control parameters of existing control connectors. Symmetrically, control parameters of the composite connector are connected to control ports of existing elements. Required data ports are connected with plain data ports that satisfy the constraints associated with the connected required data ports. The semantics of the connection (denoted by thin dotted lines in the figure) is that of identity. The data connectors adjacent to the required data port inside the composite connector *identify* the required port with its connected port, i.e. the plain data port becomes the source or target of the data flow defined within the composite connector.

4.2 Construction

In order to build up composite connectors from the set of basic connectors and data coordinators (see Figure 3), we need ways to compose them. In this section, we first describe such composition mechanisms. Further, we show how composite connectors can be composed hierarchically.

Composing basic connectors and data coordinators

There are three mechanisms for composing basic connectors and data coordinators in our model: (i) aggregation, (ii) composition via data ports, and (iii) composition via control ports.

Aggregation is the way of composition in which control- and data flows defined by the composed connectors are simply aggregated (their union is created), without any interaction between the flows. Aggregation can be used to compose a group of data connectors or a group of data and control connectors. An example of the former is shown in Figure 6. Each of the two FIFO channels connects a required output port to a required input port of their parent connector. The parent connector defines data flow from the sources a and b to the sinks c and d. For an example of the latter, i.e. a control connector and a data connector composed within a composite connector with no interaction between their flows, see the Strategy-TR connector in Figure 7(a). The FIFO channel going from *data* to the lower port of the data switch is aggregated with the selector control connector.

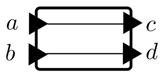


Figure 6: Aggregation

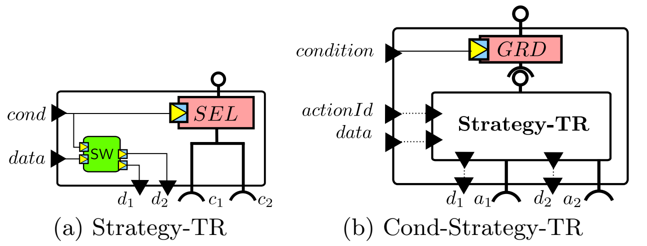


Figure 7: Example Controller Patterns

Composition via data ports composes data flow defined by a data connector and either control flow or data flow defined by an entity with a data port. Such entity can be either a control connector with data input (selector, guard) or a data coordinator. In the former case, data flow and control flow interact by a data connector feeding the data to a control connector that needs it for its control routing decisions. In Figure 7(a), we can see a FIFO channel going from *cond* to the data port of the selector connector. In the latter case, the resulting connector defines a dynamic data flow pattern whose behaviour depends on the semantics of a chosen data coordinator. In Figure 7(a), the data switch coordinator is composed with four FIFO channels via its data ports. It directs the data flow coming through the channel from *data* to one of the channels connected to d_1 or d_2, based on the data coming from *cond*.

Composition via control ports allows for the composition of control connectors, and thus control flows they define. Each control connector defines a basic control flow pattern coordinating its control parameters. To compose two control connectors, one connector (child) becomes a control parameter of another one (parent), by connecting the control port of the child with the control parameter of the parent. The control flow of the resulting composite connector is obtained by replacing the part of the flow corresponding to the control parameter of the parent connector with the control flow defined by the child connector. Figure 8 depicts the general schema for composing control connectors. The dashed lines denote control flows.

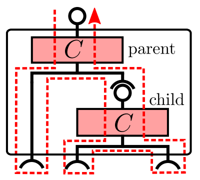


Figure 8: Control Flow Composition

Hierarchical construction

Complex coordination patterns, including controller patterns, often consist of other simpler patterns that are useful in their own right. To enable the construction of such composite patterns as well as to handle the complexity of connector design, our composite connectors can be composed in a hierarchical fashion.

Composite connectors share some of their interface elements, namely control port and control parameters, with control connectors. These can, therefore, be composed with other elements inside a parent composite connector in the ways described earlier in this section. For example, see the control port of a Strategy-TR instance composed with the control parameter of the guard connector in Figure 7(b). Required data ports can be linked to the data ports of data coordinators, or they can be transitively linked to the required data ports of their parent connector. The latter case means the parameters (constraints on plain data ports) of the inner connector template are propagated as parameters of the new connector template.

5. A COMPLETE EXAMPLE

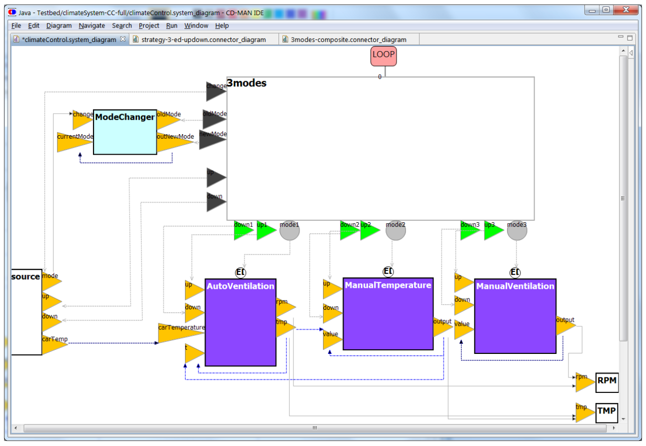
In this section, we demonstrate the usage of controller patterns for the construction of reactive control systems. We present a case study of a climate control system [8, 11] implemented in our prototype tool. The main focus remains on designing the controller of the system, composed out of simpler controller patterns.

5.1 Climate Control System

The climate control system is a reactive software system controlling the climate in a car. The user controls the climate in a car by means of a control panel with three buttons (Mode, Up, Down). In addition to the input coming from the panel, the system also receives the current temperature from a thermometer inside the car. The climate is controlled by two actuators, a ventilation fan and a car heater, by the system outputting the fan's speed and the desired temperature of the heater. When active, the system operates continuously in a feedback control loop, its actions influencing the following measurements.

The system is an instance of a common class of systems with modes. It operates in one of three modes: AutoVentilation, ManualTemperature and ManualVentilation. The modes are changed cyclically in the given order by the user pressing the Mode button. In each mode, the system's strategy for computing its outputs and interpretation of inputs is different. In the AutoVentilation mode, the system adjusts both controlled variables to achieve and maintain the set temperature (changed by Up and Down buttons). In ManualTemperature, the system simply sets the temperature of the heater and keeps it regardless of the real temperature in the car. Similarly, in ManualVentilation, the systems only sets the fan's speed (changed by Up and Down).

As observed by Labbani et al. [8], the architecture of a system with modes can be factored to a processing part, comprising a component per mode that computes the system functionality in that mode, and a coordination part that deals with mode switching. We showed how such separation can be achieved in our model without composite connectors [11]. The architecture for the climate control system contained source and sink components for data exchange with the environment, three E/D components performing computations corresponding to the three modes, a data flow component computing the current mode from the previous mode and user input, and a network of data and control connectors and data coordinators. Here, we present a refactored version that uses controller patterns for mode switching. Figure 9 shows the system architecture with the controller functionality encapsulated in the 3-Modes controller pattern.



Java - Testbed/climateSystem-CC-full/climateControl.system_diagram - CD-MAN IDE
File Edit Diagram Navigate Search Project Run Window Help
*climateControl.system_diagram
strategy-3-ed-updown.connector_diagram
3modes-composite.connector_diagram
LOOP
3modes
ModeChanger
change
currentMode
oldMode
outNewMode
source
mode
up
down
carTemp
AutoVentilation
ManualTemperature
ManualVentilation
carTemperature
rpm
tmp
value
output
RPM
TMP

Figure 9: The Climate System Architecture

5.2 Controller Design

The controller coordinates the connected components by means of both control and data flow. The 3-Modes connector's design is detailed in Figure 10. We can see that if there

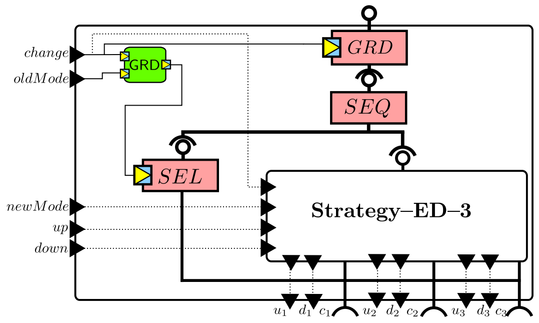


Figure 10: 3-Modes Controller Pattern

is any change in mode (detected by the guard control connector), the control flow firstly disables the component corresponding to the old mode (the selector connector performs the choice according to the *oldMode* input). Consequently, the sequencer delegates the job to another controller pattern, called Strategy-ED-3. Notice that control parameters of 3-Modes are shared between the selector and Strategy-ED-3. Data entering the connector are used as conditions for control flow routing (*change* and *oldMode*) or are being handled by Strategy-ED-3.

Strategy-ED-3 (see Figure 11) is used for enabling the E/D component that corresponds to the new mode. It also makes sure that only the currently active component (there is always exactly one) is fed by inputs (information about user pressing the Up and Down buttons). Notice

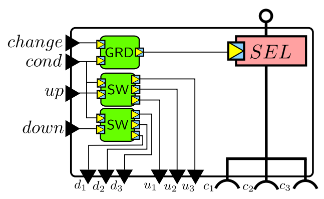


Figure 11: Strategy-ED-3

that the control is sent to a component only if the modes are changed ($change = true$) and the component needs to be (de)activated, while the data is being routed to the inputs continuously, matching the data-driven semantics of an E/D component.

It is worth noticing that 3-Modes and Strategy-ED-3 do not have direct dependencies on particular components, which stems from the coordination nature of control and data connectors in our model. On the other hand, the interfaces of both connectors have some required data ports that correspond to data ports of processing components (*up* and *down*). Similarly, the fact that our system operates in three modes is hard-wired into both connectors. This is, indeed, an unnecessary limitation, whose elimination is planned as future work.

6. DISCUSSION AND CONCLUSION

Controller patterns, represented as composite connectors in our component model, provide an abstraction suitable for constructing controllers in reactive software systems.

They allow semantic distinction between controllers and processing components to be made explicit on the architectural level, in line with architectural patterns that argue for such separation, such as the Recursive Control pattern in Figure 2. Unlike those patterns, however, the controller patterns are precisely defined in the context of the component model and can be directly used in the construction of control software.

Related component models based on port connection cannot make such a separation due to the nature of their connectors, which are either implicit in the case of ADL-like component models and Scade, or they are not sufficiently expressive (not a Turing complete set of control connectors) and not fully composable in the case of ProCom. Controllers thus end up being defined as components.

However, component representations of controllers may entail dependencies not inherent in the controller's logic imposed by the properties of a component model, such as the dependency of required ports on their interfaces in ADL-like models. Unlike these, our controller patterns depend only on control signals and data routed through a controller, i.e. dependencies inherent in the controller's logic. The loose coupling between controller patterns and processing components consequently increases their reuse potential.

The same principles underlie a class of coordination languages [5] that separate coordinators and coordinatees. Compared to them, our component model seems to be original in explicitly expressing both control and data flow. Control-driven models, such as X-MAN [9], lack explicit data flow representation in the architecture; whilst data-driven models, such as Reo [1], lack explicit control flow.

For further future research, we consider introducing *roles* to raise the level of abstraction of composite connector interfaces. The roles would aggregate the required ports and control parameters that logically belong to one entity, and express additional constraints on them. Roles could also be parametrised with multiplicities, and they could be potentially used for decreasing the number of identity connections needed to be drawn manually.

Another possible avenue of research is to explore the usefulness of composite connectors in product-line software engineering. The composite connectors might play the role of a flexible skeleton of the reference architecture, into which various components could be plugged to derive a product-specific architecture.

To conclude, controller patterns presented in this paper are a testimony of the ability of the component-based software development to express reusable abstractions, often already existing in the form of patterns [4], and make them tangible, supported by tools for their creation, storing in the repository and instantiation in a system. This strengthens our belief that patterns do not have to be only pieces of text but they can become first-class entities in component-based software construction.

7. REFERENCES

[1] F. Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, 2004.

[2] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A classification framework for software component models. *IEEE Trans. Soft. Eng*, 37(5):593–615, 2011.

[3] Esterel Technologies. *SCADE Language Reference Manual*, 2010. http://www.esterel-technologies.com.

[4] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. *Design Patterns.* Addison-Wesley, 1995.

[5] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, Feb. 1992.

[6] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and models of concurrent systems*, pages 477–498, New York, NY, USA, 1985.

[7] P. Haslum. Patterns in reactive programs. In *Proc. of the Cognitive Robotics Workshop*, 2004.

[8] O. Labbani, J.-L. Dekeyser, and P. Boulet. Mode-automata based methodology for Scade. In *LNCS*, volume 3414, pages 386–401. Springer Verlag, 2005.

[9] K.-K. Lau, M. Ornaghi, and Z. Wang. A Software Component Model and Its Preliminary Formalisation. In *FMCO'06*, pages 1–21. Springer-Verlag, 2006.

[10] K.-K. Lau and T. Rana. A taxonomy of software composition mechanisms. In *EUROMICRO-SEAA*, pages 102–110. IEEE, 2010.

[11] K.-K. Lau, L. Safie, P. Štěpán, and C. Tran. A component model that is both control-driven and data-driven. In *CBSE'11*, pages 41–50. ACM, 2011.

[12] K.-K. Lau and Z. Wang. Software Component Models. *IEEE Trans. Soft. Eng.*, 33(10):709–724, 2007.

[13] D. Lea. Design patterns for avionics control systems, Jan. 17 1995.

[14] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *ICSE'00*, pages 178–187. ACM Press, June 2000.

[15] B. Selic. An architectural pattern for real-time control software, Oct. 17 1996.

[16] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In *CBSE'08*, pages 310–317. Springer, 2008.

[17] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.