

A Component Model that is both Control-driven and Data-driven

Kung-Kiu Lau, Lily Safie, Petr Štěpán and Cuong Tran
School of Computer Science
The University of Manchester
Oxford Road, Manchester M13 9PL, United Kingdom
kung-kiu,safiel,stepanp,trancc@cs.man.ac.uk

ABSTRACT

In some industrial domains, it is beneficial to model systems that are both data-driven and control-driven. The challenge to component-based development (CBD) is to provide suitable component models for this purpose. In this paper we propose such a component model. We define the model, and present a model-driven implementation. We also illustrate its application to an example from the automotive domain.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques; D.2.11 [Software Engineering]: Software Architectures

General Terms

Design

Keywords

data flow, control flow, component model, separation of control and data

1. INTRODUCTION

In general, every software system is made up of three basic elements: (i) control; (ii) computation; and (iii) data. Computation means executing function evaluation, assignment, etc., and data is the set of values used in computations and the values that result from computations. Control is used to trigger these executions.

The order of computation executions can be defined in different ways, depending on whether the computations are *control-driven* or *data-driven*. In the classic von Neumann architecture, computations are *control-driven*. Executions occur one at a time, and control defines the order of computation executions; control flow can be sequencing, branching or looping. By contrast, in data flow architectures, computations are *data-driven* (as in data flow languages [12]). Any

number of computation executions can occur simultaneously at any one time, and there is no control. Computation executions are triggered by data flow, i.e. a computation executes when all its required data is available. Consequently, the order of computation executions is non-deterministic.

In industry, tools and languages for developing systems in different domains are designed according to the domain's needs. They can be predominantly control-driven (e.g. Esterel [4] and Argos [22]) or data-driven (e.g. Lustre [7] and Signal [20]), or more usually they are a mixture of both. For CBD, the challenge is to define component models that can provide component-based counterparts to these tools and languages.

In this paper, we consider the design of a component model that is both control-driven and data-driven, but has separate control flow and data flow. For the domain of embedded systems, it is desirable to separate control from data flow. For example, the standard tool for the avionics domain,¹ SCADE (Safety Critical Application Development Environment) [9], separates control from data flow, with control defined by Esterel and data flow defined by Lustre. However, SCADE is only data-driven, but not control-driven since it defines and uses control signals as data values as input to its data flow. In contrast, in the component model we propose, we want to not only separate control flow and data flow, but also make the model both control-driven and data-driven. Being control-driven would make our model more expressive, in terms of defining control flow, i.e. in terms of building control in a structured manner. Being data-driven would enable our model to emulate SCADE.

The contribution of our work is twofold. Firstly, we defined such a control-driven and data-driven component model with an explicit separation of control flow and data flow. Secondly, we developed a model-driven implementation in order to demonstrate the practicality of the new component model.

2. RELATED WORK

In this section, we briefly survey related work, in the form of current component models that have elements of control flow and data flow, but with varying degrees of separation between them. We will consider all three categories that current component models fall into [19, 17]: (i) models with objects as components, e.g. EJB [8]; (ii) models with architectural units as components, e.g. SOFA [5] and Palladio [2]; and (iii) models with encapsulated components, e.g.

¹This domain requires compliance with the DO-178B standard.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CBSE'11, June 20–24, 2011, Boulder, Colorado, USA.
Copyright 2011 ACM 978-1-4503-0723-9/11/06 ...\$10.00.

X-MAN [19, 16]. We will show whether they are control-driven and/or data-driven.

In component models where components are objects, computation is purely control-driven, and occurs when an object calls the method of another object. Control flow is thus the sequence of method calls. Like control, data is passed from the caller to the callee and returned by the callee to the caller; so data flow coincides with control flow.

The only component model with encapsulated components is X-MAN [19, 16]. In X-MAN computations are completely control-driven. Data flows together with control flow.

In component models where components are architectural units, control flow and data flow can be easily separated by components having distinct ports for control and data, and their associated connectors. However, this separation alone does not determine whether the computations are control-driven or data-driven. The latter depends on the semantics of the component model. For example, in ProCom² [23, 6], a component has a control port and a data port and connectors for control ports and for data ports (Figure 1).

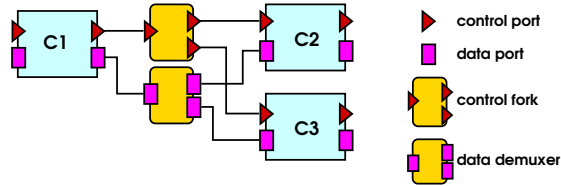


Figure 1: ProCom components and connectors

However, despite this separation, a ProCom system is strictly control-driven, and not at all data-driven. In a component, data flow is control-driven, i.e. it is triggered by control signals on the control ports.

In SCADE, the underlying component model uses architectural units as components. It separates control from data flow and is data-driven, but not control-driven. SCADE combines the data flow programming language Lustre [10] and the imperative language Esterel [4]. Figure 2, taken from [14], shows a Climate Control system in SCADE.³ It shows how the separation of control and data flow is achieved by having a state machine defined in Esterel to model the control part and a Lustre block to model the data flow part. Note, however that in SCADE control flow cannot be defined. Rather, state machines output control signals, and these signals are used as data input to the Lustre blocks. Thus SCADE is purely data-driven.

In our proposed model, we want to achieve the separation of control and data flow as in SCADE [15]. In addition, we want to be able to define control flow properly, i.e. we want to be able to define control structures and not just individual control signals like in SCADE. In other words, unlike SCADE, we want our component model to be truly both control-driven and data-driven.

3. OUR COMPONENT MODEL

In this section, we give the definition of our proposed component model. To make the model data-driven, we define

²ProCom is a successor of SaveCCM [11].

³A detailed description of a similar system can be found in Section 6.

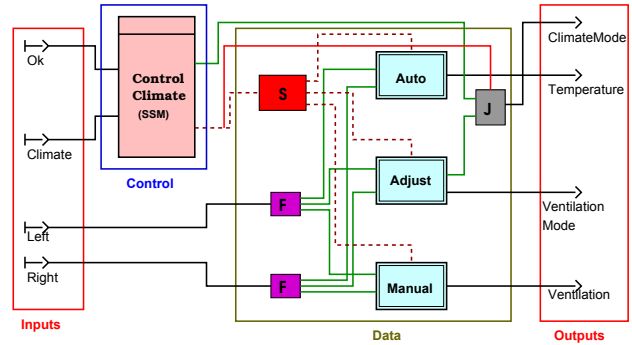


Figure 2: Separation of control and data in SCADE

components that are purely data flow units of computation, as in data flow programming languages [12]; and define connectors for these components as data channels. To make the model control-driven, we also define components that are data flow units of computation that also have control ports; and define connectors for these components as control structures that coordinate control flow between the components.

3.1 Components

Components are units of data or computation, with input and output ports (for simplicity but without loss of generality, we will only show examples with one input port and one output port). There are three categories of components: (i) source and sink components; (ii) pure data flow components; and (iii) hybrid components.

3.1.1 Source and Sink Components

Source components are data sources and sink components are data sinks. The input and output ports of these component are *data* ports. A source component does not have any input data port while a sink component does not have any output data port. These special types of components model the environment that a system interacts with via data exchange.

3.1.2 Pure Data Flow Components

A pure data flow component is an independent unit of computation that consists of a data transformation function that transforms input data to output data. Such a component has only data ports: input data ports and output data ports (Figure 3).

The execution of the function inside the component is data-driven, i.e. it can only take place when all the input data ports are filled up with data. The processing of the data causes the input data ports to be available again for any new data. The execution of the function is done in three steps: reading the input data at the input ports, transforming the input to output by data transformation and writing the output data to all the output ports at the same time. If new data arrives at the input port during this process, it has to wait for the current func-

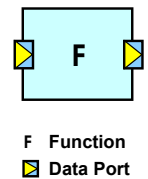


Figure 3: A pure data flow component

tion execution to complete, before being transferred into the input ports.

A real-world example of a pure data flow component is a fuel gauge on a car’s dashboard. It takes fuel-level readings from a sensor in the fuel tank and transforms the fuel level in centimetres to a percentage. The component operates all the time as long as fuel-level readings are available.

3.1.3 Hybrid Components

Hybrid components are pure data flow components but with the addition of *control* ports. A control port is activated when control flow reaches it. The execution of a hybrid component’s data transformation function is both data-driven (like a pure data flow component) and control-driven. There are two types of hybrid components: *E/D components* and *TR components*. E/D stands for Enable/Disable, while TR stands for Trigger. These components have states, which change when control flow reaches their control ports. An E/D component (Figure 4(a)) has states {Enabled, Disabled}, and its behaviour is described by the state transition diagram in Figure 4(b).

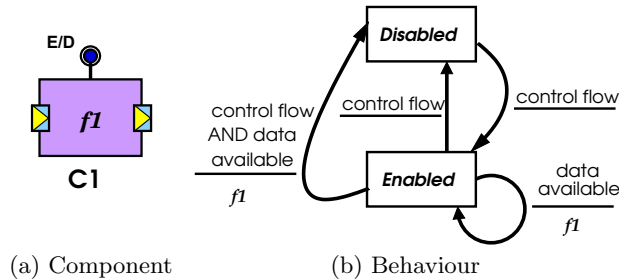


Figure 4: An E/D component

An E/D component switches between these two states when its control port is activated by control, and when in the Enabled state it will execute its data transformation function whenever data is available at all its input data ports. If control is available at the same time as data when a component is in the Enabled state, the data transformation function executes first, then the component switches to the Disabled state.

This type of a component can model, for example, a cruise controller in a car. When switched on, it adjusts the throttle value, according to incoming data from a speed sensor. The computation continues as long as input data is available, until the cruise controller is switched off.

A TR component (Figure 5(a)) has only one state {Passive}, and its behaviour is described by the state transition diagram in Figure 5(b). A TR component is always in the Passive state; however, when its control port is activated by control, it executes its data transformation, but only if data is available at all its input data ports.

An example of a TR component is the comfort function in a car that stores the driver’s ID on the ignition key, and stores the driver’s preferences in memory. When the driver inserts the ignition key, the preferred settings for mirrors, driver’s seat, etc., for the driver are read from memory. When the driver turns the ignition key, the comfort function is triggered and is executed immediately since all the necessary data is already available.

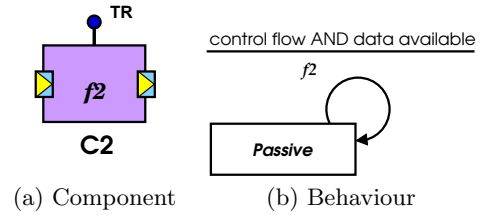


Figure 5: A TR component

3.2 Connectors

Connectors are used to establish connections between two ports. There are two main types of connectors in our model: (i) data connectors and (ii) control connectors; for connecting data and control ports respectively. In addition, we also have data coordinators that coordinate data flow between data connectors.

3.2.1 Data Connectors

A *data connector* is a directed edge which connects two data ports – an output data port to an input data port. Connected data ports must have compatible types. A data connector is a pipe that transfers data from its source (output data port) to its sink (input data port). Whenever a datum is produced at the output data port, it is transferred by the data connector to the input data port (that it is connected to).

We have three main types of data connectors. They are defined using the semantics of data channels in the REO coordination language [1]; their REO names are given in brackets for reference. We only briefly explain them here.

1. **Unbounded FIFO** – This type of data connector has an unbounded FIFO (first in first out) buffer between the source and the sink. This means that it can store an unlimited number of data items in the buffer. An unbounded FIFO queue enables the connected entities to work independently, consuming and producing data asynchronously. (*FIFO*)
2. **Size one FIFO** – The behaviour of this type of connector is similar to the unbounded FIFO except that the FIFO queue for the data connector has a buffer of size one. Such a connector might incur data loss when a new value from the source replaces the value in the buffer that is waiting to be consumed by the entity at the sink. (*Shift-lossy FIFO1*)
3. **Non-destructive read size one buffer** – In contrast to a FIFO channel that deletes the value in the channel once it is copied to the sink (component input port), the non-destructive read size one buffer does not delete the data value when it is copied to the input data port at its sink. The data value remains in the buffer until the new data arrives at the source and the current data value is overwritten with the new value. (*Variable*)

Data connectors behave according to their semantics regardless of the types or states of components to which they are connected. For instance, a FIFO data channel connected to a disabled E/D component still stores all incoming data in its buffer. The stored data will be processed by the E/D

component after it is enabled. Different behaviour can be achieved by changing the data connector. For example, a size one FIFO channel, which does not buffer old data, is suitable for processing only the most up-to-date data.

In principle, one data input port can be connected to multiple data output ports (and vice versa). However, this implies non-deterministic choice of which data is written to the input data port if multiple data items arrive at the same time. As a result, data loss may occur. To prevent unintended data loss it suffices to avoid such many-to-one connections, e.g. by using a data switch (see Section 3.2.2) that allows for deterministic data coordination.

3.2.2 Data Coordinators

Data coordinators coordinate data flow between data connectors. A data coordinator forwards data, according to some condition. It has input data ports and output data ports that are connected to data connectors. We have defined two types of data coordinators for our model: *data switch* and *data guard*.

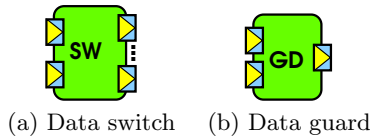


Figure 6: Data coordinators

1. **Data switch** – This coordinator has two data input ports, of which one is connected to a source that provides decision data, and the other is connected to another source which provides the data that needs to be forwarded. A data switch has multiple output data ports but only one of them will be selected to be filled with data. The selection is based on the data condition that it receives.
2. **Data guard** – It has a similar behaviour to a data switch except that it only has one data output port. It only forwards the data to the output port if the data condition is met.

3.2.3 Control Connectors

Control connectors define control flow in our model. A control connector has a control port and can connect an arbitrary number of control ports (of hybrid components or of other control connectors); it is depicted in Figure 7. A

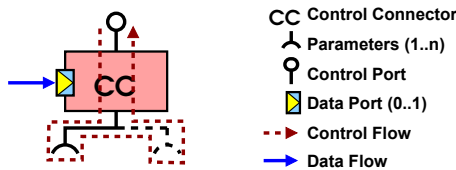


Figure 7: A generic control connector

control connector defines control flow as follows: it receives control through its control port, then passes it to the control ports it connects, and finally passes control back out through its control port. Thus a control connector *encapsulates* control.

A control connector may have a data input port; this provides data for control flow routing among the connected ports (see later).

Control connectors can connect hybrid components. More interestingly, control connectors can connect other control connectors to form complex control flows. This is illustrated in Figure 8 where the control flows of two binary control connectors, CC1 and CC2, are composed to yield a more complicated control structure. However, control flow cycles in the control structure are not allowed, i.e. control connectors form a directed acyclic graph.⁴

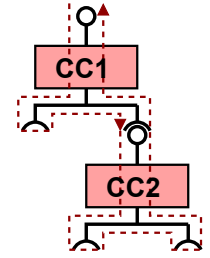


Figure 8: Composing control flows

We have defined four basic control connectors in our model: (i) Sequencer, (ii) Selector, (iii) Guard and (iv) Loop. They correspond to elementary control flow constructs: sequencing, (conditional) branching and looping. Sequencer, Selector and Guard are composable, whilst Loop is not.

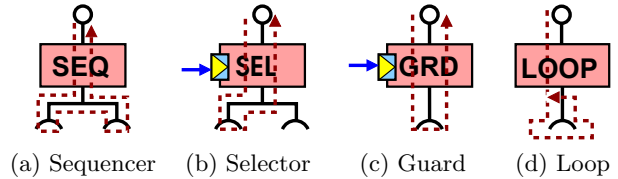


Figure 9: Control connectors

1. **Sequencer** – A sequencer routes control to its connected ports in sequence. An example of a binary sequencer is shown in Figure 9(a) together with the direction of control flow.
2. **Selector** – A selector (see Figure 9(b)) routes control flow to only one of its connected ports selected on the basis of a data input (an integer denoting the position of the port to be selected). A selector thus imposes a branching behaviour for the control flow. Selectors require synchronisation since they are points where control flow and data flow meet. Control flow stops and waits in a selector until data is available at a selector’s data input port.
3. **Guard** - A guard (see Figure 9(c)) is a unary connector that conditionally forwards control to its only connected port. It needs a data port that supplies the boolean condition. Like in a selector, control stops and waits in a guard until its data input is available.
4. **Loop** – A loop is also a unary connector. A Loop connector represents an infinite loop that keeps routing control to its only connected port. Unlike other control connectors, a Loop connector never returns control via its port; rather, after receiving control, it repeatedly passes control to its only connected port. Therefore, A Loop can only be used at the top-level of a system that runs perpetually.

⁴The control structure is detailed in [18].

4. EXECUTION SEMANTICS

In this section we describe the execution semantics of our model. We show how we can build a system using our components and connectors, and explain how the system works. We will use a simple example as an illustration.

Using our component model we can build two kinds of systems: (i) systems that are purely data-driven; (ii) systems that are both data-driven and control-driven.

4.1 Purely Data-driven Systems

To build a system that is purely data-driven, we use pure data flow components, together with data connectors. To model the environment for the system, some of the components should be source components and some of them should be sink components, which together model the data exchanged between the system and the environment, i.e. the input and output data of the system. Figure 10 shows an example of a purely data-driven system.

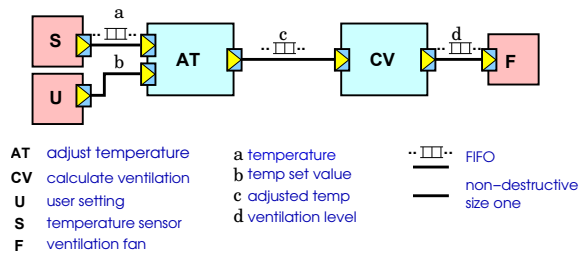


Figure 10: A purely data-driven system

The example is a room temperature controller that continuously controls the temperature in a room by means of a ventilation fan. The user can set a value for the desired temperature and the controller will work the fan to achieve it. There are two inputs to the system: a temperature reading from the temperature sensor (S) and a desired temperature value set by the user (U). The system produces one output which is the ventilation level for ventilation fan (F). S and U are the source components for the system, and F is the sink component for the system. The Adjust Temperature component (AT) is used to perform the comparison of temperature set by the user and the real temperature readings from the room (from S). The output from the comparison is used to decide the level of ventilation required for controlling the temperature according to the user setting. This is the function of Calculate Ventilation component (CV). CV will output the command to adjust the ventilation level for F. The components are connected by data connectors that are FIFO channels and non-destructive read channels.

For a purely data-driven system, the execution semantics is that of the ‘pipe and filter’ architectural style. The components are the filters and the data connectors are the pipes that model the data flow. The data-driven semantics of data flow programming languages is implemented by a scheduler that executes the data transformation function in every component whenever the data is available at the input ports of the component. There can be at most one data value at an input or output data port. When all the input data ports for a component are filled with data values, then the component can start executing the function by reading the data values from all the input ports. The input data ports will be empty again where the next available data that is queued

in the pipe of the data connector is transferred to fill up the input data ports. The data output are produced and written to all the output ports of the component at the same time when execution of the function inside the component completed. Once the output data is written at the output ports, the component is ready to execute again provided all the data are available at the input ports.

4.2 Data-and-Control-Driven Systems

To construct a system that is both data-driven and control-driven, we use hybrid components (as well as pure data flow components, and source and sink components) together with data connectors and control connectors. Hybrid components are used so that we can control the data transformation operations via the enabling or disabling, and triggering at the control port of such a component.

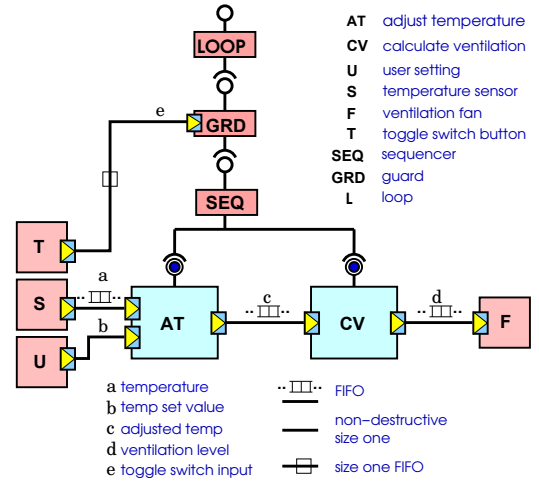


Figure 11: A data-and-control-driven system

Figure 11 shows an example of a system that is both data-driven and control-driven. It is a similar room temperature controller except that it has a feature of reducing the energy consumption by turning off the ventilation fan. The system has an additional input from button T (modelled as a source component T) to turn the ventilation fan on or off (we call this input toggle switch). We replace the data flow components AT and CV (in the previous example) with E/D components. We add a sequencer to do the enabling/disabling of AT and CV in a sequential order. A guard connector is connected to the top of the sequencer so that control flow will only be forwarded to the sequencer whenever there is an input from button T. The data port of the guard connector is connected to the button T via a data connector which is a size one FIFO.

In comparison to the previous example, the temperature controller can only work when components AT and CV are enabled. The enabling and disabling of these components are the result of actions from toggle switch button T.

For a system that is both data-driven and control-driven, the execution semantics involves both data flow and control flow. These two flows drive the execution of a system by executing different elements of the component model. The data-driven execution is implemented by a data flow scheduler that performs the execution of the pure data flow components, the data connectors and the E/D components (with

the condition that the state for E/D component is enabled and the data is available at the input data ports).

The control-driven execution is implemented by a control flow thread that executes the control flow defined by the control structure of the system. Control flow is responsible for executing control connectors, TR components and switching the state of E/D components. A control flow thread is spawned once at system start-up and enters the top-most connector in the connector hierarchy. Depending on the connector type, the control flow either keeps iterating the top-most Loop, or it terminates after one traversal of the control structure if the top-most connector is not a Loop.

The existence of two flows running in parallel brings synchronisation problems since some information is accessed by both. For example, the state of E/D components is changed by control flow but is also accessed by the data flow scheduler. When no synchronisation is imposed, the order in which an E/D component's state is changed and the scheduling of its execution is unspecified and varies non-deterministically every time a system is run. In general, this can result in unpredictable system behaviour. In the example in Figure 11, when AT is enabled and the system receives the batch of input data in which the T button is pressed, there are two possible system behaviours: either the sensor readings S and U are processed by AT if it is not disabled prior to being scheduled for execution; or the readings are unprocessed if AT is disabled and not scheduled for execution. In this example, the practical consequence of this asynchronicity may be negligible, but in general it is a serious problem that needs to be addressed, since it results in ambiguous execution semantics.

As a solution, we adopted the *synchronous execution model* [3]. The model assumes (i) infinitely fast computation and (ii) infinitely fast data transfer. Essentially, this means that the actual time taken for computation and data transfer does not matter as they do not overlap. The system execution is a series of cycles in which computation and data transfer happen in sequence.

Figure 12 depicts the adaptation of the synchronous execution model for our model. In addition to data transfer phase, each cycle comprises the scheduling of data-driven components (i.e. pure data flow components and E/D components), their parallel execution and one control flow iteration. By ensuring that execution scheduling and control thread iteration are sequenced we avoid the aforementioned problem.

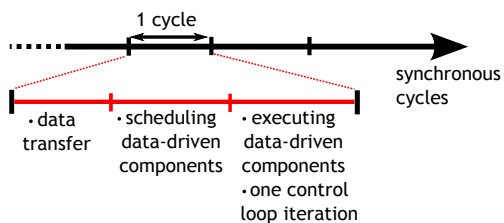


Figure 12: Synchronising data flow and control flow

In fact, this execution model allows us to execute also data-driven systems, and therefore we use it for all systems built using our component model.

5. A MODEL-DRIVEN IMPLEMENTATION

In order to build systems using our model we have implemented a prototype tool. The tool is comprised of two parts: a system architecture editor and a system simulator.

The system architecture editor provides a system developer with a graphical environment to create the architecture of a system. Figure 16 shows a screenshot of the editor. A developer can instantiate individual components, compose them using data and control connectors and configure their attributes. The editor has been developed using model-driven engineering techniques, i.e. it has been generated by Eclipse Graphical Modeling Framework (GMF) from its meta-model.

The meta-model (a simplified version) can be seen in Figure 13. It captures the entities from the component model definition. The main entity is **System** that contains all components, connectors, sources and sinks comprising the system. The **Component**, **ControlConnector**, **DataConnectorOperator** and **DataConnector** entities are further subclassed by their particular semantic variants. For instance, **Component** has **DataComponent** and **HybridComponent** subclasses corresponding to pure data flow components and hybrid components, respectively. Figure 13 does not show the attributes of meta-model classes. The attributes are used to fully specify the behaviour of meta-model entities, e.g. specification of a component's function or run-time semantics of a hybrid component (E/D or TR).

The system simulator is integrated with the modeller part of the tooling (through the Eclipse platform and its plug-in mechanism). Its function is to simulate the execution of systems designed in the system modeller. It takes a system's specification and simulator configuration⁵ and runs the system in accordance with the model's execution semantics. A simulator configuration associates sources and sinks in the system specification with input and output files. It also contains some initialisation data, e.g. the initial states of E/D components or initial contents of some channels. The simulator's output is the files associated with the sinks as well as a trace of system execution for debugging purposes.

In order to implement the simulator, we needed to create the run-time counterparts for meta-model entities realising the semantics prescribed by the component model definition. For instance, a component, when executed, invokes a Java class specified in the system architecture that performs the component's computation. Hybrid components have to have a state (enabled/disabled) and have to react to incoming control based on their semantics (E/D or TR) and current state. Data ports also have a state telling whether they are full or empty. Likewise, data connectors' semantics had to be implemented (FIFOs, non-destructive read channels), etc.

Another significant part of the simulator is a realisation of the execution semantics defined in Section 4. When the simulator is started, it spawns a thread for control flow and another thread for the data flow scheduler, which is running a dataflow loop (Figure 14). In each iteration of the loop, the data is transferred from the components' output ports to connected channels and from channels to connected input ports if they are empty (the `channelsMoveData()` method). Next, in accordance with the data flow semantics, the set of

⁵Currently, the simulator configuration is specified directly in the system's specification in the architecture editor.

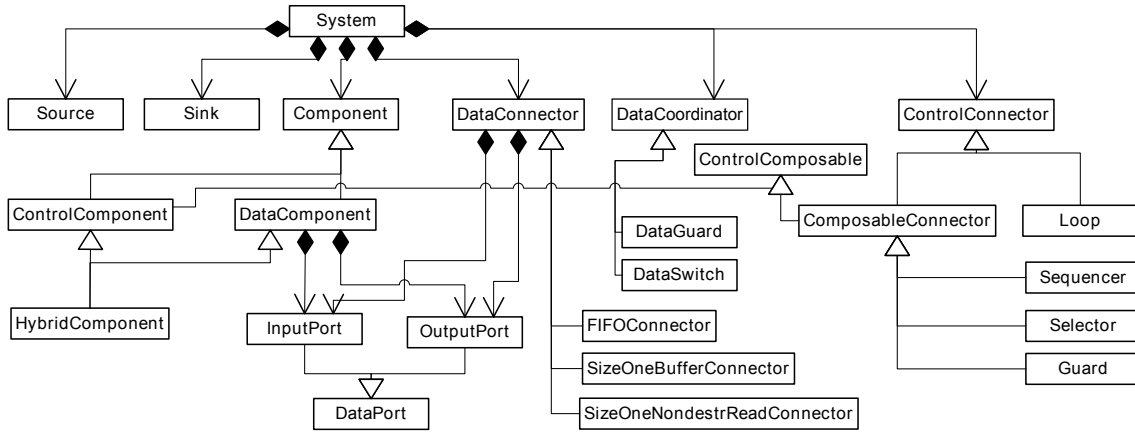


Figure 13: Metamodel

```

while(running) {
  channelsMoveData();
  componentsToRun = getReadyComponents();
  clockTick(); //sync with control th.
  executeComponents(componentsToRun);
  waitForControl(); //sync with control th.
}

```

Figure 14: Dataflow loop

ready-to-execute components (pure data flow and enabled E/D components) is determined and the components are executed in parallel (the `executeComponents()` method). The loop is executed until the user terminates the simulator. The control thread iteratively goes through the connector hierarchy. The two threads (data flow scheduler and control thread) are running concurrently and are synchronised according to Section 4. One period of synchronous execution is delimited by two consecutive calls of the `clockTick()` method. It is a signal for the control thread to make one iteration of the top-most Loop⁶. The `waitForControl()` method waits for the control thread to finish that iteration.

6. EXAMPLE

In this section, we demonstrate the construction of a system in our model using the prototype tool. We use the case study of a simplified version of climate control system used in cars that has been adapted from a similar case study presented in [14] (Figure 2 in Section 2). We show how the separation of control flow and data flow in modelling the system can be achieved using our model.

6.1 Climate Control System

A climate control system is used in cars to regulate the temperature and the ventilation mode. It is controlled by means of a simple device depicted in Figure 15(a). There are three buttons, Mode, Up and Down, and two displays showing the current value of temperature (in Centigrade) and the current speed of the ventilation fan (in rotations per minute), two variables controlled by the system. The

⁶The current prototype only supports a system having a Loop as the top-most connector.

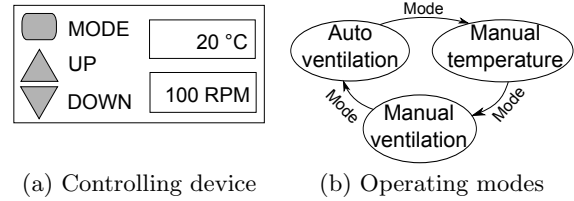


Figure 15: Climate control system

device operates in three modes, which are changed by pressing the Mode button. In the **Auto Ventilation** mode, a user increases or decreases the current temperature by one degree using Up and Down buttons, respectively. The fan speed is set automatically based on the actual temperature in the car (measured by a sensor) and the current temperature setting. In the **Manual Temperature** mode, pressing Up and Down buttons changes the current temperature in the same way as in the previous mode but the fan speed cannot be changed. Likewise, in the **Manual Ventilation** mode, Up and Down regulate the speed of the ventilation fans, not changing the temperature setting.

In our case study, we construct a software system controlling this device. The system takes four inputs: three booleans (`true` when a button is pressed, `false` otherwise) for Mode, Up and Down buttons, and an integer representing the actual temperature in the car (needed by the **AutoVentilation mode**) measured by a sensor. The system outputs just two values: temperature and ventilation fan speed. The system is sent inputs periodically and produces outputs in response, with the same frequency. Once started, the system enters the **Auto Ventilation** mode.

6.2 System Architecture

Our system is a typical example of an embedded system with running modes. The behaviour of the device's buttons is dependent on the mode which is being changed at runtime. In other words, the change of mode *controls* the system's behaviour. When the system is in a particular mode, it is *driven by data* – incoming Up and Down button clicks and sensor readings. Therefore, the system possesses character-

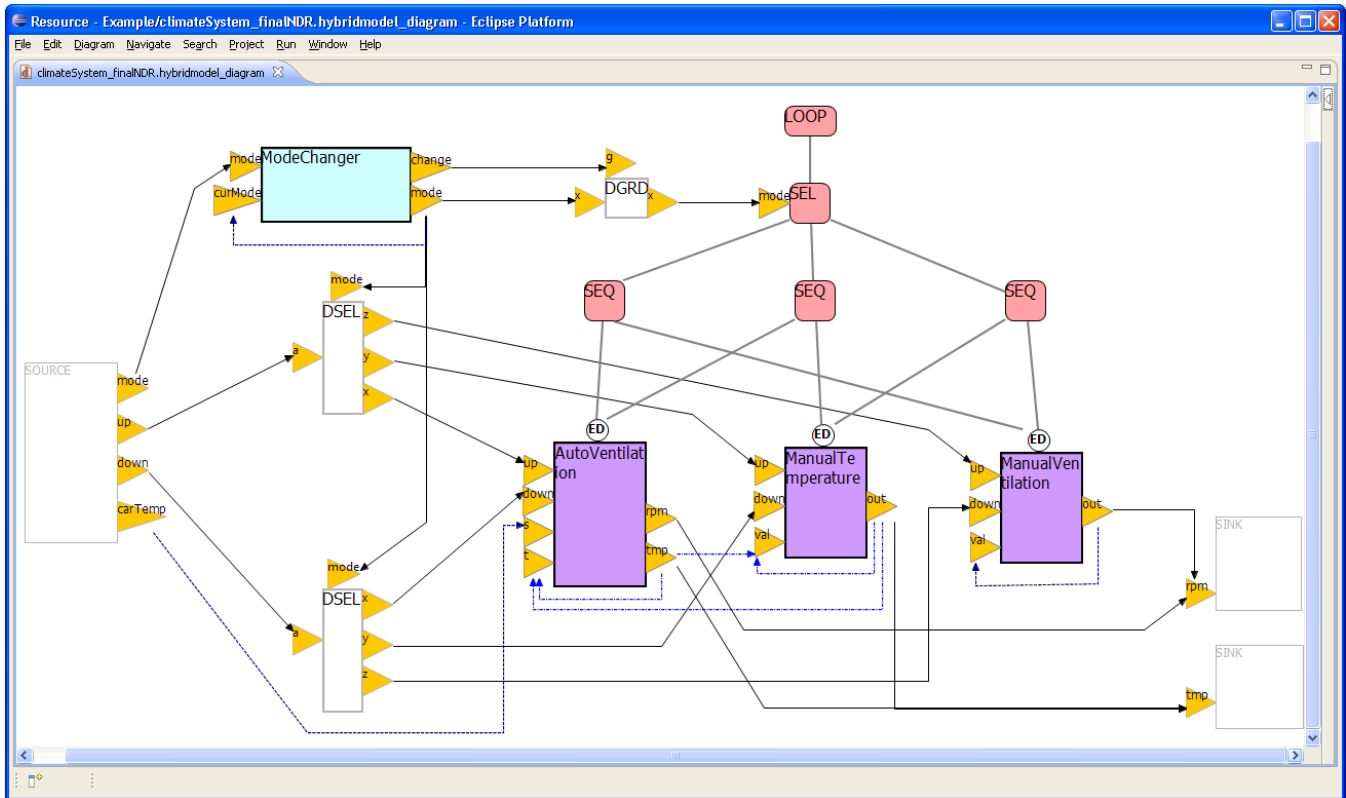


Figure 16: Climate system architecture in the Architecture editor

istics of both control- and data-driven architectures, which can be separated by means of our model.

Figure 16 shows the architecture of the climate control system created using our component model. On the left, there is a source feeding input data into the system. The source's output ports produce data at the same time periodically. On the right, there are two sinks. One is reading temperature settings produced by the system, and the other one is reading the ventilation fan speed. In the middle are three E/D components corresponding to the modes of the control device and a hierarchy of control connectors that allows for mode switching. Additionally, there is a dataflow component **ModeChanger**. It plays a role in changing the current mode of the system.

At any time, exactly one of the E/D components **AutoVentilation**, **ManualTemperature** and **ManualVentilation** is active and it processes incoming data (button presses and in case of **AutoVentilation**, sensor readings) and computes outputs values (one or both depending on the mode). Initially, it is **AutoVentilation**. Later in the system execution, when a user changes mode by pressing the Mode button, control connectors deactivate the component performing the current mode's computation and activate the component corresponding to the new mode. Information about the Mode button state (a boolean value) is first processed by **ModeChanger**, a dataflow component that based on the button's state and the current mode determines whether a system should switch to a new mode and if so, it sends the new mode to the Selector connector. If the system should remain in the same state, the Selector connector is not sent any data. This filtering functionality is performed by a data

guard (**DGRD** entity in the figure). When control thread comes to Selector and finds the data ready on its port, it is forwarded to one of the Selector's children (mode is an integer in $[0,2]$ and is used as index in an array of the children) – three Sequencer connectors. On arrival of control, they redirect it *in sequence* to two of the above-mentioned E/D components, deactivating one and activating the other, respectively.

To achieve synchronisation between data inputs coming to the system at the same time there are several *data switches* in the system architecture. The data switches redirect the data coming from their input channels to one of their output data channels according to the current mode (sent from **ModeChanger**) and thus guarantee that the Up and Down buttons' presses will be processed in the right mode (the mode that has been selected by preceding input data batch in which *mode = true*).

The example architecture in Figure 16 uses three types of data connectors: unbounded FIFO (black full line), size one FIFO (dash-and-dotted line) and non-destructive read size one buffer (dashed line). Non-destructive read size one buffers in this example effectively simulate state for components **ModeChanger** and **ManualVentilation** and supply the most up-to-date sensor readings to **AutoVentilation**.

Finally, note how the control part of the system represented by control connector hierarchy is explicit and separate from the dataflow part of the system represented by dataflow and E/D components. The two parts are then connected and synchronised together by means of data connectors and data stream operators (data switches and guards).

Input				Output		
<i>mode</i>	<i>up</i>	<i>down</i>	<i>carTemp</i>	Mode	<i>rpm</i>	<i>tmp</i>
False	True	False	10	AV	1000	21
False	True	False	10	AV	1000	22
True	False	False	10	MT	–	22
False	False	True	10	MT	–	21
True	False	False	10	MV	0	–
False	True	False	10	MV	1	–
False	True	False	10	MV	2	–

Table 1: Expected behaviour of the climate control system for our test scenario

6.3 System simulation

Here, we briefly illustrate how our prototype tool can be used for simulating system execution. As mentioned in Section 5, we need a system architecture and simulator configuration to run the simulator. We use our climate system and create the following configuration:

Configuration item	Value
Data connector <i>tmp</i> → <i>t</i>	20
Data connector <i>mode</i> → <i>curmode</i>	0
Data connector <i>ManualVentilation.out</i> → <i>curmode</i>	0
AutoVentilation.active	<i>true</i>

We have omitted the specification of input and output files for the source and sinks. The configuration describes the initial state of the system: initial contents of some data connectors and the state of the E/D component responsible for the initial mode. Values stored in data connectors correspond to the initial temperature setting, mode and fan speed setting, respectively.

Further, we need to create an input file which will be read by the system’s Source. In this test scenario, we model the following sequence of buttons presses: *Up* → *Up* → *Mode* → *Down* → *Mode* → *Up* → *Up*, and a constant sensor measurement of 10°C for simplicity. Table 1 shows the expected behaviour of our climate control system with this specific data input. Each row in the Input column corresponds to one batch of inputs produced by Source at one time. The Mode column contains the current mode. The values in the Output column show expected values to be computed by the system for the inputs presented on the same row in the table.

When we execute the simulator, it shows the results in a window for immediate inspection (see Figure 17). Comparing Figure 17 and Table 1, we see that the system has produced expected results and therefore passed the test.⁷

7. DISCUSSION AND CONCLUSION

It is acknowledged that separation of control and data in software architecture results in a number of benefits [15]. As an instance of the separation of concerns principle, it leads to increased reuse, modularity and more structured architecture, which in turn is reflected in an increased understandability. Another possible area that could take advantage of the separation is verification. Different parts of the system

⁷The values for Mode are not included in the simulator’s output since the current mode was not sent to any sink.

Source				Sink 1	Sink 2
<i>mode</i>	<i>up</i>	<i>down</i>	<i>carTemp</i>	<i>rpm</i>	<i>tmp</i>
false	true	false	10	1000	21
false	true	false	10	1000	22
true	false	false	10	0	22
false	false	true	10	1	21
true	false	false	10	2	
false	true	false	10		
false	true	false	10		

Figure 17: Simulator output for our test scenario

could be verified separately and, due to lesser complexity, more efficiently than the whole.

So far, these benefits have not been achieved fully by a single component model, and this motivates us to propose a new component model. Like ProCom, our component model allows for the separation between control flow and data flow by means of separating model entities that form control flow (i.e. control connectors) and data flow (i.e. data connectors).

Like SCADE, our model allows for the clear distinction of control and data processing parts in the system architecture. Comparing Figures 2 and 16, we see that the same partitioning of a system architecture into parts dealing with input/output, control and data processing present in SCADE (Figure 2) can be achieved in our model as well: sinks and sources deal with input/output; control connectors with control; and data connectors and components with data (flow and processing). This cannot be achieved in models with control-driven execution semantics, such as ProCom, which (much like a procedure call) forces control and data flow to be bound together.

Unlike SCADE, control flow is explicit in our model since it can be modelled in system architecture using control connectors. In SCADE, control flow is implicit in the form of state machines [15]. Furthermore, in our model, arbitrarily complex control flows can be constructed by composing control connectors. The composition of control connectors corresponds to a control connector defining more complex flow. The connector composition is hierarchical, giving a clear control structure.

The set of component model entities presented in this paper is not final. We are still in the process of refining the model definition. Possible additions of new entities include other data connectors (REO [1] provides a rich source of various channel semantics), data coordinators (various operators from data flow process networks [21], such as data selector, delay) and control connectors (COBEGIN for adding concurrency, finite loop). Additionally, we are considering enhancing component definition. One possible extension would be adding state to components. A more challenging enhancement is introducing the notion of composite components. It is possible to build composite components from data flow components and data connectors by means of port delegation, as is the current practice in ADLs. However, defining composite components combining both control flow and data flow is much more involved. We intend to explore this area in our future research.

Further, we plan to run a series of case studies in various

domains to extend the evaluation of the component model. Potentially, the ability of our model to model control flow and data flow could be exploited for the analysis of domain requirements, as an alternative to the Feature Oriented Domain Analysis (FODA) [13] methodology. Control connectors would seem to correspond to control transformations in FODA, and pure data flow components would seem to correspond to primitive data transformations; and therefore a system built using our model would correspond to a DFD (data flow diagram), which is functional model within the domain model. Thus, it seems plausible that our component model can be used to define domain models in a component-based manner.

Finally, although we have achieved the goal of building systems that are both control-driven and data-driven in a component-based manner, it is too early to fully evaluate our approach in terms of practical benefits. However having a tool will enable us to test our approach on more realistic applications, which in turn will point out further refinements that we need to make to the component model.

8. ACKNOWLEDGMENTS

We would like to thank anonymous reviewers who provided us with valuable suggestions on improvement of the paper.

9. REFERENCES

- [1] F. Arbab. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Comp. Sci.*, 14(3):329–366, 2004.
- [2] S. Becker, H. Koziol, and R. Reussner. The Palladio Component Model for Model-driven Performance Prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009.
- [3] G. Berry and L. Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, 1985. Springer-Verlag.
- [4] F. Boussinot and R. de Simone. The ESTEREL language. *Proceedings of the IEEE*, 79(9):1293 –1304, Sept. 1991.
- [5] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proc. SERA '06*, pages 40 –48, Aug. 2006.
- [6] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. Vulgarakis. ProCom – the Progress Component Model Reference Manual, version 1.0. Technical Report MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.
- [7] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A Declarative Language for Programming Synchronous Systems. In *14th ACM Conf. on Principles of Programming Languages*, pages 178–188, 1987.
- [8] L. DeMichiel and M. Keith. Enterprise Javabeans 3.0 specifications, 2006.
- [9] Esterel Technologies. *SCADE Language Reference Manual*, 2004. <http://www.esterel-technologies.com>.
- [10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proc. of the IEEE*, pages 1305–1320, 1991.
- [11] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren. SaveCCM – A Component Model for Safety-Critical Real-Time Systems. In *Proc. 30th EUROMICRO SEAA*, pages 627–635. IEEE, 2004.
- [12] W. M. Johnston, J. R. P. Hanna, Richard, and J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, 2004.
- [13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
- [14] O. Labbani, J.-L. Dekeyser, and P. Boulet. Mode-automata based methodology for Scade. In *Proc. 8th Int. Workshop HSCC*, volume 3414, pages 386–401. Springer Verlag, 2005.
- [15] O. Labbani, J.-L. Dekeyser, and É. Rutten. Separating Control and Data Flow: Methodology and Automotive System Case Study. Research Report RR-5832, INRIA, 2006. <http://hal.inria.fr/inria-00070193/PDF/RR-5832.pdf>.
- [16] 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.
- [17] K.-K. Lau and T. Rana. A Taxonomy of Software Composition Mechanisms. In *Proc. 36th EUROMICRO SEAA*, pages 102–110. IEEE, 2010.
- [18] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *CBSE'05*, pages 90–106, 2005.
- [19] K.-K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, October 2007.
- [20] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. Signal-A data flow-oriented language for signal processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 34(2):362 – 374, Apr. 1986.
- [21] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.
- [22] F. Marainchi and Y. Rémond. Argos: an Automaton-Based Synchronous Language. *Computer Languages*, (27):61–92, 2001.
- [23] 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.