Domain-Specific Software Component Models

Kung-Kiu Lau and Faris M. Taweel

School of Computer Science , The University of Manchester
Manchester M13 9PL, United Kingdom
{Kung-Kiu,Faris.Taweel}@cs.manchester.ac.uk

Abstract. We believe that for developing applications in a specific domain, the best kind of software component model to use is a domain-specific one. We also believe that current component models intended for specific domains are actually not domain-specific. In this paper we present an approach for deriving domain-specific component models from the domain model of a given domain, and show why such a component model is better than existing models that are not domain-specific.

1 Introduction

The usefulness of software components for constructing systems is recognised in increasingly diverse problem domains. General-purpose software component models [11,17] like EJB [7] and architecture description languages (ADLs) [19] are well-established for CBSE in generic problem domains. More recently, the principles of CBSE have been adopted by specialised problem domains like embedded systems [9], and even hardware design [24].

We believe that for a specialised domain, the best kind of component model is a domain-specific one. Moreover, we believe that such a model should be derived from the *domain model* [12]. By and large, this is not the case for current component models that are meant for special domains, e.g. Koala [22] for consumer electronics, PECOS [20] for field devices and SaveCCM [1] for vehicular systems. These models tend to be generic ADLs. They take account of the domain by incorporating its context into architectural units.Their connectors remain generic and simply link the ports of the architectural units. We do not therefore, regard these models as domain-specific because they are not derived directly from the underlying domain model. We believe that a domain-specific component model should have not only components that are domain specific, but also composition operators that are domain-specific, following the definition of component models in [11,17].

In this paper, we present an approach for deriving a component model from a domain model. In general, a domain model consists of many sub-models, such as the functional model [12,10] and the feature model [6]. The functional model describes data and control flow in the domain, starting from data and control I/O to the domain, via intermediate data processing operations, to the most basic data processing functions. The feature model specifies the functional units and their inter-relationships, in the domain, as well as which are mandatory or optional.

A component model consists of components and composition connectors. In this paper, we show how we can derive a domain-specific component model from a domain

G.A. Lewis, I. Poernomo, and C. Hofmeister (Eds.): CBSE 2009, LNCS 5582, pp. 19–35, 2009.
© Springer-Verlag Berlin Heidelberg 2009

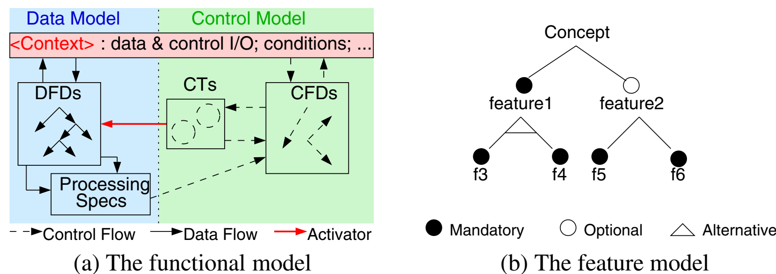
model. We derive domain-specific components from the most basic data processing functions, and we derive domain-specific composition operators from the control flow specification, in the functional model (of the domain model). A domain-specific component model not only makes it easier to build applications, by the use of pre-defined components and composition operators, but it can also take into account important domain knowledge such as variability, which generic ADLS cannot do.

2 Domain Models

There are various definitions of what a *domain* is. In this paper, we follow Czarnecki's definition [6]: "an area of knowledge scoped to maximise the satisfaction of the requirements of stakeholders, which includes concepts and terminology understood by practitioners in that area and the knowledge of how to build (parts of) systems in the area". We loosely interpret a domain as a problem domain.

A domain is described in a *domain model*, which is defined in a *context* described by a context model. A context model defines interactions between the candidate domain and external domains, together with any external constraints on these interactions [12]. In addition to the context model, the domain model consists of many other (sub)models (Fig. 1), each describing a different aspect (or view) of the domain. For example, the *feature model* (Fig. 1(b)) represents the common and variable features of a concept instance (an application in the domain), together with the dependencies among variable features. Other (sub)models include the *functional* model, *object* model, the *entity-relationship* model, etc. Domain modelling is a well-established research area, and has a more or less standard terminology. We will follow [8,10,12] for terminology.

In this work, we focus on the *functional* and *feature* models. The functional model describes the *data (or structural) model* and the *control model* (Fig. 1(a)). In Fig. 1(a), starting from the context (top), the functional model specifies functional structure and behaviour in terms of a *data model* (left) and a *control model* (right) respectively. The data model describes data flow in the domain, starting from data I/O to the domain (specified in the context), as a hierarchical decomposition of functionality in the domain where each node is a *data flow diagram* (DFD). Within a DFD (Fig. 2), functions are specified as *data transformations* (dt's) interconnected by data flows (solid lines). The decomposition process of data ends with *primitive* data transformations (pdt's). Each primitive data transformation is associated with a specification for data access or



(a) The functional model (b) The feature model

Fig. 1. Sub-models of a domain model

general data processing. The control model reuses the same hierarchy of DFDs of the data model, but without the data flows. It instead shows control flows (dashed arrows) for data transformations, and is captured in *control flow diagrams* (CFDs). Each CFD can be associated with a *control transformation* (*ct*),[1] which receives input control signals form its CFD, processes or transforms them, outputs them, and activates or deactivates data transformations in the corresponding DFD (activation/deactivation is represented by open arrows as in Fig. 1(a)). For example, in Fig. 2, *ct2* inputs control flows that are handled by the primitive data transformations *pdt2* and *pdt3*.

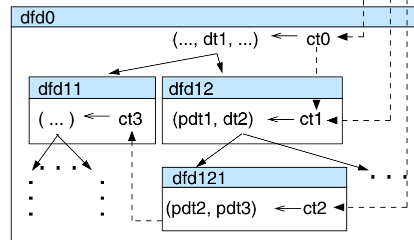


Fig. 2. Hierarchical decomposition of DFDs

A control transformation is a specification for a finite-state machine (FSM). A pair of corresponding DFD and CFD forms a *decomposition level* which may or may not have a control transformation. We introduce a visual simplification here embedding each *ct* in the relevant DFD and dropping CFDs, and control flows from the CFDs are represented as arrows arriving directly at *ct*'s.

As a result of tests on input data flows by their primitive data transformations, control flows are generated and propagates back to control transformations at the same or different decomposition levels. In particular, primitive data transformations can feed control signals back to an arbitrary control transformation, e.g. in Fig. 2, *pdt3* may feed control back to *ct2*.

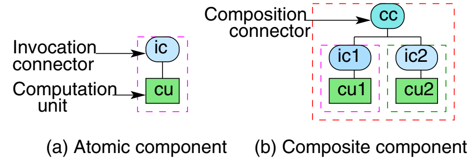
Finally, for domain-specific development approaches such as generative programming [6] and product-lines engineering [4], a domain model provides the starting point. In these approaches, the domain model is used to build architectures and domain-specific languages. An architecture must capture variability so as to represent product families [23], and domain-specific languages are used to define such families and generate individual products. In generative programming, for example, an architecture is represented as templates based on GenVoca grammar [3], and code for different products can be generated from instances of such templates. In this paper we propose an alternative approach. From a domain model, we derive a domain-specific component model. This model encapsulates domain knowledge in its components and composition operators. These can be deposited in a repository for the domain, and reused for building any product (in the domain). Therefore our approach obviates the need for designing (and implementing) both architectures and domain-specific languages. Rather our domain-specific component model combines both.

3 Our Component Model

Our approach to deriving domain-specific component models is based on a component model that we have defined for a generic domain [16,14]. In this section, we briefly describe the component model which we refer to as the generic (component) model.

[1] In Fig. 1(a), CTs are put outside DFDs, but henceforth we put them inside for clarity.

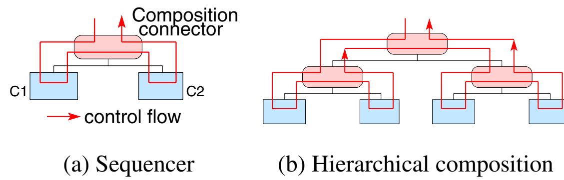
The generic model has two kinds of basic entities: (i) *computation units*, and (ii) *connectors* (Fig. 3). A computation unit *CU* encapsulates *computation*. It provides a set of methods (or services). Encapsulation means that *CU*'s methods do not call methods in other computation units; rather, when invoked, all its computation occurs in *CU*. Thus *CU* could be thought of as a class that does not call methods in other classes.



(a) Atomic component (b) Composite component

Fig. 3. Our component model

There are two kinds of connectors: (i) *invocation*, and (ii) *composition* (Fig. 3). An invocation connector provides access to the methods (and data) of a computation unit. A composition connector encapsulates *control*. It defines and coordinates control flow between a set of components (atomic or composite). For example, a *sequencer* that composes components $C_1, \ldots, C_n$ can call methods in $C_1, \ldots, C_n$. This is illustrated in Fig. 4(a) for two atomic components. The control encapsulated by the composition connector determines the control flow between the two components.



(a) Sequencer (b) Hierarchical composition

Fig. 4. Composition in our model

Another example is a *selector* composition connector, which selects (according to some specified condition) one of the components it composes, and calls its methods.

Components are defined in terms of computation units and connectors. There are two kinds of components: (i) *atomic*, and (ii) *composite* (Fig. 3). An Atomic component consists of a computation unit with an invocation connector that provides an interface to the component. A composite component consists of a set of components (atomic or composite) composed by a composition connector. The composition connector provides an interface to the composite.

Invocation and composition connectors form a hierarchy [16]. This means that composition is done in a hierarchical manner. Furthermore, each composition preserves encapsulation. This kind of compositionality is the distinguishing feature of the generic component model. An atomic component encapsulates computation, namely the computation encapsulated by its computation unit. A composite component encapsulates computation and control. The computation it encapsulates is that encapsulated in its sub-components; the control it encapsulates is that encapsulated by its composition connector. In a composite, the encapsulation in the sub-components is preserved. Indeed, the hierarchical nature of the connectors means that composite components are self-similar to their sub-components, i.e. composites have the same structure as their sub-components; this property provides a basis for hierarchical composition.

Fig. 4(b) illustrates this: it shows that a generic composition connector receives control, passes it to its sub-connectors, and returns control. The control it receives comes

from a composition connector at the next level of composition, i.e. the next level up, and the control it returns goes to a composition connector at the next level up too.

Finally, the generic model defines only *sequential* systems, i.e. systems that execute only one process, sequentially. It is also worth noting that components are *passive*, and executes only when invoked.

4 Deriving a Domain-Specific Component Model

To derive a domain-specific component model, we examine the domain model, and identify data transformations as candidates for atomic or composite components, and control transformations as candidates for composition connectors. Clearly to turn these into elements of the component model, we need to map or transform them into the latter. In this section, we discuss how this can be done.

4.1 Atomic Components

First we need to adapt atomic components in our model, in order to interpret control transformations according to our model. As we have seen in Fig. 2, a decomposition level in the functional model of a domain model comprises a DFD, a CFD and possibly a control transformation. A control transformation transforms a control flow from one type to another and activates or deactivates data transformations. In general, activation starts an *active* component executing, and deactivation stops it executing. Active components have their own thread of execution (unlike passive components, e.g. those in our component model), and are commonly elements of reactive or concurrent systems [2]. Since our component model does not have concurrency, we have to interpret activation and deactivation as method invocation in a (passive) atomic component. An invocation may succeed or fail, depending on whether the component has been activated or not. Therefore we need to add a *guard* to an atomic component; this is an adaptor connector that guards control flow to the computation unit by storing the state of a flag for the activated/deactivated status. The behaviour of a guard is shown in Fig. 5(b), where the two control flows show activation and deactivation as successful and failed invocation respectively. For simplicity, we will not show the guard in atomic components explicitly in any figures after Fig. 5(b).

With this adaptation, we can identify atomic components in a given functional model. Every primitive data transformation (*pdt*) is mapped to an atomic component, with its own invocation connector and a guard connector. A group of *pdt*s associated with a control transformation (*ct*) is also mapped to an atomic component if the *ct* activates only one *pdt*, and all the other associated *pdt*s return only data values, but no control signals at all. Fig. 5(a) shows an example of this. In this case, all the *pdt*s (pdt_1, pdt_2) become one atomic component with a guard, as in Fig. 5(b).

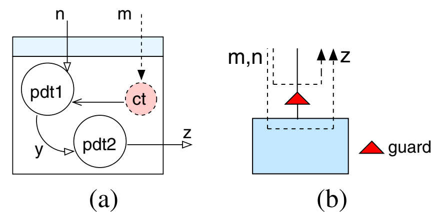


Fig. 5. Atomic components

In general, there can be many additional *pdt*s that are not part of the *ct* specification. These can be aggregated around *pdt*s that are specified in the *ct*, as in Fig. 5(a).

In the figures, we use variables to denote data values that accompany control flow; we use $m, n, p, \ldots$ to represent input values, and $x, y, z \ldots$ to represent output values for a connector. For example, in Fig. 5(b), m, n is the input to the guard (and the invocation connector which is not shown), and z the output from it.

4.2 Composition Connectors

In general, a *ct* is a control coordinator; a property that it shares with composition connectors of our component model (Section 3). However, unlike our composition connectors (Fig. 4), *ct*s do not encapsulate control. At any decomposition level of the functional model, a *pdt* may produce *feedback control flow*, i.e. it may output a control signal. In particular, such feedback control flow may ultimately arrive at a *ct* at another decomposition level, bypassing the *ct* at its own level.

Our strategy for deriving composition connectors from *ct*s is to identify groups of *ct*s which together do encapsulate control (as in Fig. 4). We call such a group a *control-encapsulation region* (*CER*). Of course, in general, for a given domain model, there are different ways to form *CER*s. However, for each *CER* we can derive a composition connector as defined in our component model. Such a connector is typically a composite connector [13] in the generic model. It is a domain-specific connector.

To determine *CER*s in a given functional model, we work up from the bottom decomposition level of the functional model. At each level, if there is a *ct*, say ct_1, then we check the control flow that starts from ct_1 and trace the control flow through the *pdt*s and to any other control transformation ct_2, and so on, until we reach the end of the control flow path. All the *ct*s in the control flow path then define a *CER*, and hence one composite composition connector in our model [13]. This composite connector encapsulates the control defined by all the control transformations $ct_1, ct_2, \ldots$ in this *CER*.

*CER*s can be broadly classified into two groups, depending on whether or not there is any feedback control flow therein. For each of these groups, we can also identify different cases, depending on the nature of the feedback control flow, if any, or depending simply on the interactions between the *pdt*s and the *ct*s. For each of these cases, we can define a corresponding composition connector in terms of connectors in the generic model. Here we briefly discuss the different categories of *CER*s we have identified so far and the corresponding composition connectors.

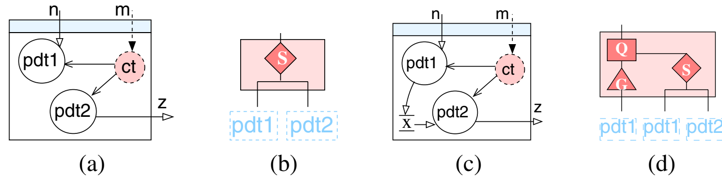


Fig. 6. No *pdt*s producing any feedback control

Category 1. A *CER* may contain no feedback control flows. A possible case is depicted in Fig. 6(a). Here the ct activates one *pdt*, but deactivates all other ones. The specification of the derived connector depends on whether these *pdt*s exchange data flows or not. A ct activating two *pdt*s that do not exchange data flows with one another maps to a *selector* composition connector, S in Fig. 6(b). However, a data flow from one to another requires a data store to allow passing the data values, and imposes a precedence on which *pdt* to perform first. The specification of the derived connector therefore must reflect these requirements for the connector to correctly achieve the desired functionality in the domain. Fig. 6(c) and (d) show the *CER* and the resulting composite connector in the case of interacting *pdt*s. The connector is a hierarchy comprising a selector (S), a sequencer (Q) and guard (G).

Category 2. A *CER* in this category contains a ct which receives feedback control flows from *pdt*s in the same decomposition level, as in Fig. 7(a). The only meaningful way to handle the feedback control flow is to activate other *pdt*s in the *CER* (and deactivating its source). Effectively, the feedback control flow is processed by the *ct* such that a sequence of one or more *pdt*s are performed next as a result of the feedback control flow. In the generic component model, this corresponds to sequencing the source *pdt* of the control flow with a guarded sequencer or pipe. Guarding is necessary to cater for scenarios where the control flow does not occur. Actually, the guard is equivalent to moving the source of the feedback control flow from the *pdt* to the control hierarchy, leaving the *pdt* with computation only. In general, to derive a connector for *CER*s in this category (e.g. Fig. 7(a)), we proceed in two phases: (i) a connector is derived by ignoring feedback control flows, but labelling their sources (Fig. 7(b)); and (ii) replacing the labelled branches by composites each consisting of a pipe/sequencer and guard connector (Fig. 7(c)).

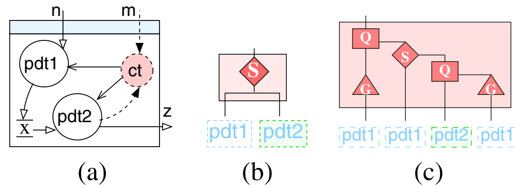


Fig. 7. Feedback control that stays within one level

Category 3. Feedback control flows can be returned by *pdt*s to cts at levels different from their own, bypassing their level's cts altogether. Such *CER*s form the third category. In Fig. 8(a), feedback control leaves one level and goes to another *ct* in the next level. A non-primitive *dt* A contains a *pdt* (pdt_1) and a non-primitive *dt* B, associated with a ct (ct_1). B is decomposed into two *pdt*s (pdt_2 and pdt_3) and is associated with its own ct (ct_2). A feedback control flow is initiated in pdt_2 and arrives (and terminates) at ct_1; one level higher in the model. The resulting *CER* therefore contains the two cts, ct_1 and ct_2. Deriving a connector for this *CER* involves ct_1 and ct_2, in addition to piping/sequencing and possibly guarding. The derivation of a connector specification from Fig. 8(a) can be achieved by (i) abstracting the details of the *dt* that generates the feedback control flow, and (ii) applying steps used for deriving connectors from *CER*s in Category 2. Applying these steps to Fig. 8(a) reduces it to Fig. 8(b) which automatically contributes the composite connector in Fig. 8(c) (similar to that in Fig. 7). The

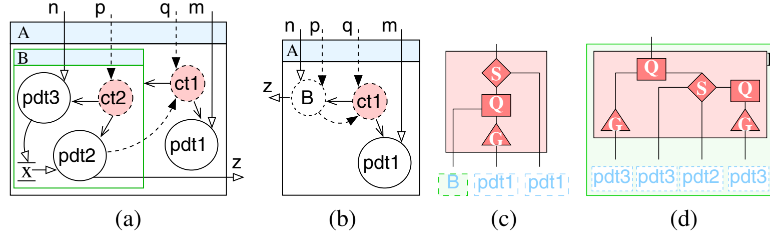


Fig. 8. Feedbacl control that goes to another level

final specification of the composite connector derived from A is achieved by replacing B in Fig. 8(c), by that in Fig. 8(d).

The composition connector derived for each *CER* is domain-specific because it encapsulates not only generic control structures that correspond to the composition connectors in our generic model, but it also encapsulates control behaviour defined in control transformations. Such control behaviour is defined in terms of state transitions between the states of the domain. Thus a derived composition connector contains domain-specific state-based behaviour.

Additionally, connectors also define place holders for control and data values, and specify how they are initialised and accessed. Data in the domain model is specified in terms of data stores defined in data transformations. Moreover, connectors are responsible for input and output data flows.

In summary, a derived composition connector is domain-specific because it specifies domain knowledge expressed in terms of control behaviour, data and data flows described in a given domain model. This will become clear in the next section when we discuss an example in some detail.

Finally, it is worth noting that in our approach, composition connectors provide an interesting means for modelling variability in domains. Based on the feature model, variants of connectors are specified to cater for optional features and their dependencies. Component models supporting domain variability specify it in terms of configuration interfaces and parametrised components, as exemplified by Koala [21]. In our approach, a set of *base* connectors are defined first, by ignoring all optional features (data transformations). A variant for a base connector is then defined, to coordinate control and data flows to an optional data transformation. In Fig. 8(a), if B is an optional data transformation, then following the same rules above, first a base connector is defined for A ignoring B (and its flows), and then a variant connector is defined from A and B. That is, we have one base connector and a variant connector.

5 Example: A Component Model for Vehicular Systems

In this section, we illustrate our strategy for deriving domain-specific component models by showing how to derive one for the vehicular systems domain (VSD). In particular, in VSD, we will identify *CER*s of categories 1, 2 and 3, as described in the previous section, and derive the corresponding composition connectors.

The domain model that we will use for VSD is a simplified (and adapted) version of an existing model for a vehicle management system presented in [10] (Chapter 26). A system in VSD is one of three possible variants. The *monitoring system* is a mandatory feature in all systems. It is used for monitoring fuel, average speed and the need for maintenance services. The *cruise control system* is a variant which allows the driver to set a value for a cruising speed that the system maintains, and enables the driver to resume to that speed at later times in the trip. The *adaptive cruise control system* is the third variant which involves an object detection functionality which allows the vehicle to automatically adapt its speed to safe levels with respect to the traffic in front of the vehicle, but when possible, allows it to return to the desired speed set by the driver during a given trip.

Fig. 9 depicts a feature model for VSD. It summarises the requirements for the three aforementioned variants. The monitoring feature, which is mandatory in all variants, will have the mandatory features *Measure Motion*, *Calibrate* and *Status*. A cruise control system will have the *Cruise* feature, whilst an adaptive cruise control system will have the *cruise* feature as well as the *Object Detection* feature, in addition. The *Object Detection* feature requires the *Cruise* feature, and Fig. 9 also shows this dependency.

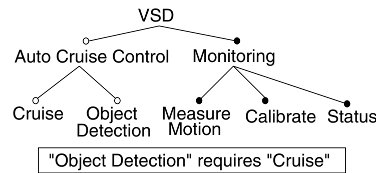


Fig. 9. Feature model for VSD

For lack of space, we only present part of the functional model. At the top level (level 0) of the functional model, we have a DFD (Fig. 10(a)), which contains one ct (ct_0) and five *dt*s: pdt_1 (1) for calculating vehicle speed and acceleration; pdt_2 (2) for calibrating the system; dt_3 ([3]) for calculating the throttle position necessary to accelerate, maintain speed at a desired speed, and optionally adapt speed to traffic in front of the vehicle; dt_4 (4) to monitor fuel levels, average speed and maintenance status; and dt_5 ([5]) for scanning obstacles, calculating safe speeds and triggering dt_3 to process the safe speed. The specification of ct_0 is a state transition diagram shown in Fig. 10(b). The diagram specifies a set of states and a number of valid transition between these states. Each transition is triggered by an input control flow to ct_0, and is then associated with a sequence of actions performed by *pdt*s in the same level. For example, when the Calibration Commands input the control signal START MEASURE to ct_0 in Fig. 10(a), this causes a

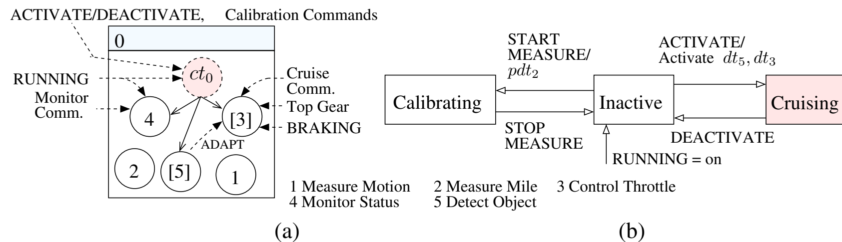
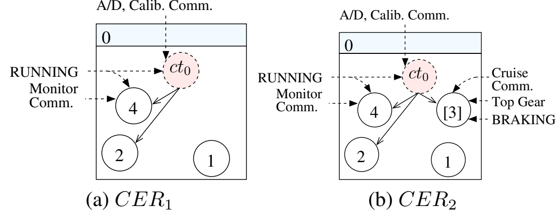


Fig. 10. Level 0 in the functional model for VSD

transition from the Inactive state to the Calibration state in Fig. 10(b). This transition activates pdt_2 as an action for handling the control signal START MEASURE.

We can identify several *CER*s in Fig. 10(a), depending on whether the optional dts (pdt_3 and pdt_5) are included or not. According to the feature model, each of pdt_3 and pdt_5 may or may not exist. If only pdt_3 exists, then we have the cruise control system; and if both pdt_3 and pdt_5 co-exist then we have the adaptive cruise control system.

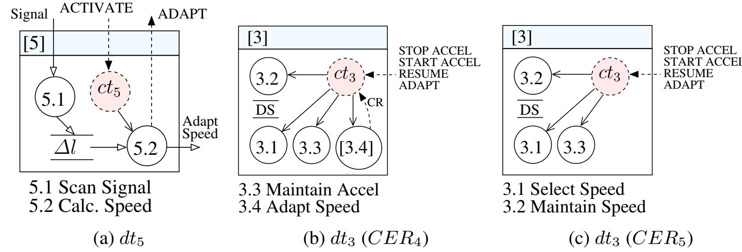


(a) CER_1 (b) CER_2

Fig. 11. Category 1 *CER*s of Level 0

Looking at Fig. 10(a), if both pdt_3 and pdt_5 are included then we can immediately define a Category 3 *CER* (let us call it CER_3) which is attributed to the control signal ADAPT returned by pdt_5 to pdt_3. If both pdt_3 and pdt_5 are dropped, then we can identify a Category 1 *CER* (CER_1 in Fig. 11(a)). If only pdt_3 is included then we have another Category 1 *CER* (CER_2 in Fig. 11(b)). We will revisit CER_3 to clarify it further below.

We can decompose the top decomposition level by decomposing the non-primitive dts (dt_3 and dt_5). dt_5 can decomposed into two primitives $pdt_{5.1}$ and $pdt_{5.2}$, together with a ct (ct_5), as shown in Fig. 12(a). $pdt_{5.1}$ analyses the Signal data flow input by a



(a) dt_5 (b) dt_3 (CER_4) (c) dt_3 (CER_5)

Fig. 12. Level 2 for dt_3 and dt_5 in the functional model for VSD

sensor (e.g. a radar) and calculates the distance separating the vehicle from the traffic in front of the vehicle. $pdt_{5.2}$ calculates a safe speed and alerts dt_3 to reduce the vehicle speed to the safe speed. ct_5 is activated together with dt_3 as a result of receiving the ACTIVATE signal, as in Fig. 10(b). Clearly, the ADAPT control flow is returned by $pdt_{5.2}$ to dt_3, thus confirming that CER_3 identified above is a Category 3 CER. Indeed, by substituting dt_5 in Fig. 10(a) by its decomposition in Fig. 12(a) we get a match with that in Fig. 8(a).

dt_3 is decomposed into four $pdts$: $pdt_{3.1}$ for recording the driver's choice of the desired speed (DS) and keeping it in a data store throughout a trip; $pdt_{3.2}$ maintains speed at the desired level; $pdt_{3.3}$ to accelerate the vehicle to a speed that a driver may wish to cruise at; and $pdt_{3.4}$ (optional) to reduce the vehicle speed to a safe speed.

These pdts are associated with ct_3, which processes a set of cruise commands (START/STOP ACCEL, RESUME, CRUISE and ADAPT). Fig. 12(b) shows the decomposition level for dt_3. The detailed behavioural specification of ct_3 is shown in Fig. 13, where the state Adapting is optional. The decomposition of dt_3 reveals two more *CER*s: a Category 2 *CER* defined by optional and mandatory pdts (CER_4 in Fig. 12(b)); and a Category 1 *CER* defined by excluding the optional $dt_{3.4}$ (CER_5 in Fig. 12(c)).

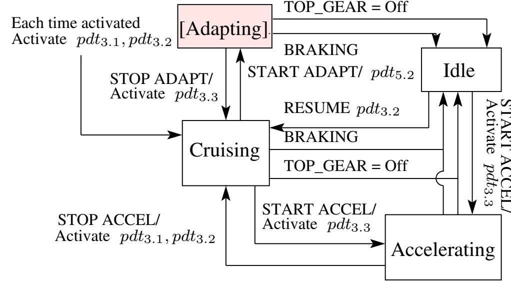


Fig. 13. Specification of ct_3

The decomposition of dt_4 is more detailed and is omitted here.[2] However, we include composition connectors derived from various decomposition levels of dt_4.

So far we have identified five *CER*s, covering all the categories presented in Section 4. Now we choose (the most interesting) three out of these *CER*s: CER_3, CER_4 and CER_5, and derive their corresponding composition connectors. In the derivation process, we start by ignoring optional dts to derive base connectors. Then, we introduce gradually the optional dts and any corresponding states in cts, and derive variant connectors for the base connector. Additionally, while working on connectors, we identify atomic components from pdts. Both, pdts and connectors obtained from the derivation process are candidates for depositing in a repository for that domain. Because composition is bottom-up in component-based development, we also derive components and connectors bottom-up.

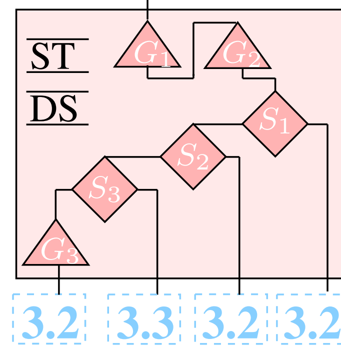


Fig. 14. CC_3

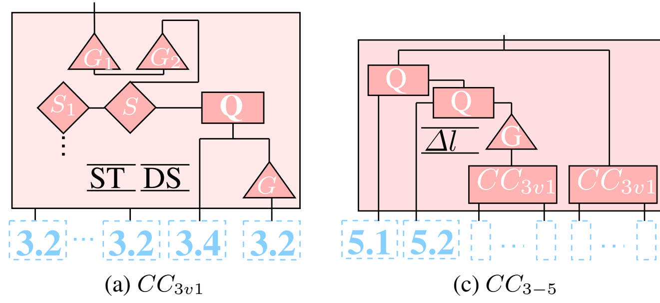
To derive a composition connector for a *CER*, we proceed as follows. We examine the state transition diagram for the ct, and map each transition to a guard or selector (in the generic model). Then we compose these connectors using our generic connectors, according to the transition conditions. The result of such a translation is a composite composition connector that is a domain-specific instance of a connector in our generic model. This is the case because of control encapsulation in a *CER*.

Consider CER_5 in Fig. 12(c) at the lowest level in the domain model. CER_5 is a member of Category 1 (Fig. 6(a)) whose pdts exchange no direct data flows. This *CER* contributes a connector (designated CC_3) whose control behaviour is defined by the specification of ct_3 in Fig. 13, excluding the optional state. Table 1 lists the connectors in CC_3's specification, the condition each connector tests for enabling a sequence of actions on pdts, and the intial states (S_f) and final states (S_i) for the transitions. (DS

[2] See the original case study in [10] (Chapter 26).

Table 1. Explanation of control processing performed by CC_3

Connector	S_i	Control Flow	DS Initialised	S_f
G_1	Accelerating OR Cruising	BRAKING	ANY	Idle
G_2	Accelerating OR Cruising	TOP_GEAR=Off	ANY	Idle
S_1	Idle	RESUME	Initialised	$pdt_{3.2}$
S_2	Cruising OR Idle	START ACCEL	ANY	$pdt_{3.3}$
S_3	Accelerating	STOP ACCEL	NOT BRAKING	$pdt_{3.1}; pdt_{3.2}$
G_3	Idle	Activate = CRUISE	ANY	$pdt_{3.1}; pdt_{3.2}$



(a) CC_{3v1} (c) CC_{3-5}

Fig. 15. Connectors derived from CER_3

is a data store, see below.) CC_3 specifies no sequences because $pdt_{3.1}$ is a pure data access operation that is performed by connectors in our model [15]. Fig. 14 shows CC_3's specification. Obviously, CC_3 is specific to VSD because its control behaviour originates from the domain model (ct_3); encapsulates and accesses two data stores: DS and the state of the connector ST; and defines interfaces ($pdt_{3.2}$ & $pdt_{3.3}$) it accepts for composition. We identify $pdt_{3.2}$ & $pdt_{3.3}$ as atomic components.

Similarly we can derive composition connectors for CER_4 (Fig. 15(a)) and CER_3 (Fig. 10). Since CER_4 is defined from pdt_3, the resulting connector (CC_{3v1}) is a variant of the base connector CC_3. Finally, CER_3 defines a connector (CC_{3-5}) based on dt_3 and dt_5, including their optional pdts and the optional state in ct_3. The connector CC_{3-5} (15(b)) is also a variant of CC_3.

5.1 Comparison with SaveCCM

It is now possible to demonstrate the advantages of our approach over other component models intended for specific domains. We have selected SaveCCM [1], a component model for vehicular systems, as a representative model. In this section, we show the design of the cruise control system (CCS) using both our domain-specific component model (Fig. 16(a)) and SaveCCM (Fig. 16(b)).

With a repository at our disposal (Fig. 17), CCS can be designed and deployed from existing connectors and atomic components. These include four connectors (CC_3, CC_4, a pipe P and a selector S); and atomic components including "Measure Mile" and "Measure Motion" (Fig. 16(a)). Two of these components ("Maintain Speed" and

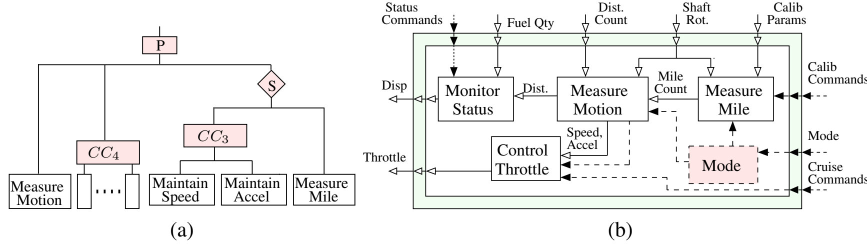


Fig. 16. Speed Controller designed using our model (a) and SaveCCM (b)

Connectors			Atomic Components	Origin
Connector	Description	Variant	Maintain Speed	$pdt_{3.2}$
CC_3	Speed controller	base	Maintain Accel	$pdt_{3.3}$
CC_{3v1}	Speed Controller	Variant of CC_3	Adapt Speed	$pdt_{5.2}$
CC_{3-5}	ACC controller	Variant of CC_3	Scan Signal	$pdt_{5.1}$
CC_4	Monitor Controller	base	Measure Mile	pdt_1
S	Selector	base	....	....
P	Pipe	base		

Fig. 17. Partial list of VSD repository

"Maintain Accel") are composed using CC_4 to construct a control cruising composite, and a number of other atomic components (not listed here) are composed by CC_4 to create the monitoring composite. CCS can either be in the calibrating mode or cruise mode, and hence "Measure Mile" and the cruising components are composed with S. The resulting composite is further composed with a pipe connector P to create CCS.

To build CCS in SaveCCM however, a set of components is selected to realise the "Cruise" and "Monitoring" features required in the domain (Fig.9). The corresponding *dt*s to these features are pdt_1, pdt_2, dt_4 for "Monitoring" and dt_3 for "Cruise" in Fig. 10. Starting with an atomic component type in SaveCCM, we create two atomic components from pdt_1 and pdt_2, and two composite components for pdt_3 and pdt_4. For example, the composite component for pdt_3 is built from three atomic components created from $pdt_{3.1}$, $pdt_{3.2}$ and $pdt_{3.3}$. Creating a composite component in SaveCCM may also require switch components. That is, each time a system is designed, we have to start from the generic types of components and connections, embed into components only domain knowledge (possibly from a domain model) and compose these components to realise the system (connections are always generic). Types in SaveCCM are not domain specific, the gap between these types and components created from them is comparable to the gap between generic programming languages and domain-specific languages [26]. Our approach delivers a set of domain-specific connectors and atomic components that comprise a repository exclusive to that domain. It is very unlikely that CC_3 can be reused in systems outside VSD. Furthermore, we have not found any literature indicating that SaveCCM components can be stored into a repository and reused.

Moreover, data access $pdt_{3.1}$ are atomic components in SaveCCM, which makes the number of sub-components in a composite large.

Having built components from dt_3 and dt_4, they can be used with pdt_1 and pdt_32 together with a switch component to construct CCS. Fig. 16(b) depicts one design for CCS using SaveCCM. The "Mode" component is required to select between performing "Measure Mile" functionality for calibrating the system and "Cruise" functionality. The composite component is constructed in two steps: (i) sub-components are connected by linking their ports to create a composite which is (ii) encapsulated by a layer (shaded circumference) specifying which of its internal ports are to be imported or exported (e.g. Throttle and Mode ports). Accordingly, the designer decides which component ports to connect and which unconnected ports to export/import. However, these two steps may not result in a component, but in an assembly. In our model, connectors are compositional and using them with atomic components always creates components. For example, P is a pipe whose control behaviour is to invoke methods in components (from left to right) passing data output by a former component's method as argument for the next component's method, and so on. The same applies to domain-specific connectors such as CC_3 which is compositional, as defined by its specification in Fig. 14 and has a pre-defined control behaviour (specified in Table 1).

Executing CCS sequentially in SaveCCM requires a Clock component to trigger the system at a fixed time period. The order of executing the CCS is not clear from the design in Fig. 16(b). Actually, to execute CCS, the clock must trigger sub-components in an order that needs to be defined. In contrast, in our model, the execution behaviour is explicit and there is always one unique point through which the system can be started: the top-level connector of the component/system (e.g. P in our design).

In Fig. 16(a), it is possible to replace the CC_3 sub-tree with another whose top-level connector is CC_{3-5} leaving other parts of the architecture untouched. This swap generates the adaptive cruise control system. This shows that our domain-specific model supports variability in terms of (variant) connectors and parametrised atomic components, which is not an intrinsic property of SaveCCM, or similar ADLs intended for specific domains.

6 Discussion

We have endeavoured to show that given a generic component model and a domain model, a new component model conforming to the semantics of the generic model can be derived from the domain model. That is, atomic components in derived models are still defined according to the rules of the generic model, and components are composed according to the composition scheme of the generic model. The resulting model is a domain-specific component model, which is main contribution of this work.

We believe that domain-specific component models are the best kind of models to use for developing applications in a given domain. This view is echoed by [18] for ADLs, and by [5,9] for embedded systems. Of course the more established field of domain engineering, including product-lines engineering [4] and generative programming [6], has always emphasised the use of domain-specific knowledge in terms of product families, and our idea of domain-specific component model is in our view a logical

extension of this field. Our contribution here is to add domain-specific component models to their armoury.

We go even further in the sense that we believe a domain-specific component model should be derived from the domain model. In this regard we are unique, as far as we know. As already mentioned, existing component models intended for specific domains are not domain-specific, according to our criteria. The main examples of these models are Koala [22], PECOS [20] and SaveCCM [1]. Although Koala is intended for the domain of consumer electronics, it does not explicitly define this domain; nor is it derived from a domain model. PECOS is intended for modelling applications in the domain of field devices. The model (component) types are generic and are derived from the control loops architectural style [25], but the domain model is not defined, and consequently any application having this architectural style could be regarded as a PECOS device. SaveCCM shares with Koala the architectural style of pipes-and-filters [25] and a Switch type. Like Koala, therefore SaveCCM is also not domain-specific.

More interesting, SaveCCM is intended for the same domain (vehicular systems) as the model we have derived in Section 5. Therefore, it provides us with a model to evaluate the feasibility of our approach. The core semantics of SaveCCM defines atomic and composite component types and conditional connections, which are all generic. Like PECOS and Koala, SaveCCM does not define the domain either. Comparing our model derived in Section 5 with SaveCCM, we see that connectors and component are specific to VSD. Whereas SaveCCM components are generic types to be used in creating domain-specific components in the design phase. Our model has more domain-specific composition connector types such as CC_3, CC_{3-5}, etc. All these connectors are unlikely to be reusable in other domains. Clearly, our approach delivers component models that reflect strong domain-specificity in terms of its types.

Unlike Koala, PECOS and SaveCCM, our approach also yields component models that support variability in the domain that is based on product families [23], a characteristic necessary for applying product-lines engineering. We claim that support of product-line engineering is an intrinsic property of domain-specific component models derived according to our approach.

7 Conclusion

Our contribution in this paper is an approach for deriving component models from the domain model. In particular, we have showed that a component model for a specialised domain can be derived from the functional model and the feature model. The resulting component model is domain-specific, not only by virtue of its types (connectors and components), but also by the fact that product families propagated into the model types via the domain. A property that these models shares with domain-specific software development approaches. We have observed that the derived models naturally support approaches such generative programming and product-line engineering.

By way of contrast, in generic ADLs for specific domains, architectural units may incorporate data transformations in general, but not specifically primary data transformations. They may also incorporate control transformations, in contrast to our approach where such transformations are in connectors, which are highly likely to be composite connectors.

References

1. Åkerholm, M., Carlson, J., Håkansson, J., Hansson, H., Nolin, M., Nolte, T., Pettersson, P.: The saveccm language reference manual. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-207/2007-1-SE, Mälardalen University (January 2007)
2. Bacon, J.: Concurrent Systems: An integrated approach to Operating Systems, Distributed Systems and Databases (Open University Edition), 3rd edn. International Computer Science Series. Addison-Wesley, Reading (2003)
3. Batory, D., O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. ACM Trans. Softw. Eng. Methodol. 1(4), 355–398 (1992)
4. Clements, P., Northrop, L.: Software product lines: practices and patterns. Addison-Wesley, Boston (2002)
5. Crnkovic, I.: Component-based approach for embedded systems. In: Proceedings of 9th International Workshop on Component-Oriented Programming, Oslo (2004)
6. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. Addison Wesley, London (2000)
7. DeMichiel, L.G. (ed.): Enterprise JavaBeans Specification, Version 2.1. Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A, November 12 (2003)
8. Gomaa, H.: Software Design Methods for Concurrent and Real-Time Systems. Addison-Wesley Longman Publishing Co., Inc., Boston (1993)
9. Hansson, H., Akerholm, M., Crnkovic, I., Torngren, M.: Saveccm - a component model for safety-critical real-time systems. In: EUROMICRO 2004: Proceedings of the 30th EUROMICRO Conference, USA, pp. 627–635. IEEE Computer Society, Los Alamitos (2004)
10. Hatley, D.J., Pirbhai, I.A.: Strategies for real-time system specification. Dorset House Publishing Co., Inc., New York (1987)
11. Heineman, G.T., Councill, W.T. (eds.): Component-based software engineering: putting the pieces together. Addison-Wesley, London (2001)
12. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University (November 1990)
13. Lau, K.-K., Ling, L., Velasco Elizondo, P., Ukis, V.: Composite connectors for composing software components. In: Lumpe, M., Vanderperren, W. (eds.) SC 2007. LNCS, vol. 4829, pp. 266–280. Springer, Heidelberg (2007)
14. Lau, K.-K., Ornaghi, M., Wang, Z.: A software component model and its preliminary formalisation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 1–21. Springer, Heidelberg (2006)
15. Lau, K.-K., Taweel, F.M.: Data encapsulation in software components. In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 1–16. Springer, Heidelberg (2007)
16. Lau, K.-K., Velasco, P.I., Wang, Z.: Exogenous connectors for components. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 90–106. Springer, Heidelberg (2005)
17. Lau, K.-K., Wang, Z.: Software component models. IEEE Trans. on Software Engineering 33(10), 709–724 (2007)
18. Medvidovic, N., Dashofy, E.M., Taylor, R.N.: Moving architectural description from under the technology lamppost. Information and Software Technology 49(1), 12–31 (2007)
19. Mehta, N.R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: ICSE 2000: Proceedings of the 22nd ICSE, pp. 178–187. ACM, New York (2000)
20. Nierstrasz, G.A.O., Ducasse, S., Wuyts, R., Black, P.M.A., Zeidler, C., Genssler, T., van den Born, R.: A component model for field devices. In: Bishop, J.M. (ed.) CD 2002. LNCS, vol. 2370, p. 200. Springer, Heidelberg (2002)

21. Ommering, R.C.: Building product populations with software components. Phd. thesis, Proefschrift Rijksuniversiteit Groningen, Met lit. opg. - Met samenvatting in het Nederlands (2004)
22. Ommering, R.C., Linden, F., Kramer, J., Magee, J.: The koala component model for consumer electronics software. IEEE Computer 33(3), 78–85 (2000)
23. Parnas, D.: On the design and development of program families. IEEE Transactions on Software Engineering SE-2(1), 1–9 (1976)
24. Preis, V., Henftling, R., Schutz, M., Marz-Rossel, S.: A reuse scenario for the vhdl-based hardware design flow. In: Proceedings EURO-DAC 1995, September 1995, pp. 464–469 (1995)
25. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
26. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Notices 35(6), 26–36 (2000)