Constructing Domain-Specific Component Frameworks through Architecture Refinement

Frédéric Loiret, Ales Pšek, Philippe Merle, Lionel Seinturier  
INRIA-Lille, Nord Europe, Project ADAM  
USTL-LIFL CNRS UMR 8022, France  
Email: firstname.lastname@inria.fr

Michal Malohlava  
Distributed Systems Research Group  
Charles University in Prague, Czech Republic  
Email: firstname.lastname@dsrg.mff.cuni.cz

Abstract—Recently, a plethora of domain-specific component frameworks (DSCF) emerges. Although the current trend emphasizes generative programming methods as cornerstones of software development, they are commonly applied in a costly, ad-hoc fashion. However, we believe that DSCFs share the same subset of concepts and patterns. In this paper we propose two contributions to DSCF development. First, we propose DomainComponents — a high-level abstraction to capture semantics of domain concepts provided by containers, and we identify patterns facilitating their implementation. Second, we develop a generic framework that automatically generates implementation of DomainComponents semantics, thus addressing domain-specific services with one unified approach. To evaluate benefits of our approach we have conducted several case studies that span different domain-specific challenges.

Keywords-component framework; generative programming; component model, domain-specific concerns

I. INTRODUCTION

Component-based Software Engineering (CBSE) [8] has emerged as a technology for the rapid assembly of flexible software systems, where the main benefits are reuse and separation of concerns. The success of this technology has been proved by variety of its applications, from the general component frameworks [4], [6], [9] to domain specific component frameworks (DSCF) addressing a wide scale of challenges — embedded [26] or real-time constraints [21], [14], dynamic adaptability [13], distribution support [25], and many others.

DSCF offers a domain-specific component model and a tool-support that allow developers to address domain-specific challenges by using appropriate abstractions available already at the component-model level. To achieve separation of concerns, domain-specific services (such as dedicated memory areas, tasks parameters, security, ...), in the literature [12] referred to also as non-functional requirements/aspects/properties/concerns, are usually deployed in the runtime platform — composed of a set of custom made containers [18].

Today, a plethora of DSCFs emerges. However, based on our experience, from a concise and specifically designed component model [23] to a full fledged component framework [21] is a long road. Although the current trend emphasizes generative programming methods [11] as cornerstone of software development, generative methods are usually tailored to specific domains and applied in a costly, ad-hoc fashion which prevents from any reuse or amelioration of solutions to a framework construction. We however believe that DSCFs share the same concepts and patterns to their construction and application.

Therefore, this paper brings two key contributions. First, we propose Domain Components — a high-level abstraction of domain-specific services provided by the container. Second, we develop a generic framework employing techniques of generative programming [11] to create custom and component-based runtime platforms leveraging development of Domain Components. Moreover, we introduce architecture optimizations independent from the target domains, which contributes to better performance of resulting applications. Finally, to evaluate our approach, we have conducted several case studies addressing domain-specific challenges and we report on the benefits acquired.

To reflect the goals, the paper is structured as follows. Section II stipulates the context of this work and clarifies more precisely our goals. In Sect. III we introduce the basic concept that we employ — Domain Components and the Generic Framework. In Sect. IV we describe HULOTTE — our prototype implementation. Section V presents the case studies that we have conducted to evaluate the proposed approach. We discuss related work in Sect. VI. Section VII concludes and draws future directions of our research.

II. CONTEXT AND GOALS

Domain-Specific Component Frameworks and their Application. Typically, DSCF is composed of a component model and the tool support which permit assembling, deploying and executing demanded applications [5]. Moreover, such component model defines the relevant architectural concepts, called domain-specific concepts, according to the requirements of the targeted application domain (e.g. to address the distribution support or real-time constraints).

A recognized methodology of developing DSCF [10] is composed of several steps as it is illustrated on Fig. 1a. In this case, each component model (step 1) is used to develop functional concerns of the application — functional components. Typically, functional components encapsulate a business logic of an application.

Afterwards, the framework tool-support is employed to create a runtime platform, in Fig. 1a step 2. The runtime platform is composed of a set of containers [18] that encapsulate functional components, and its goal is to relieve the developer from dealing with domain-specific...
requirements and to implement the execution support. Current trend in developing the runtime platform emphasizes a generative programming approach. While this task can be seen only as an engineering challenge, the runtime platform plays a crucial role in deciding whether the component model itself will be successful in real-life applications, since its implementation has a direct impact on the performance of a given application. Here, different optimizations should be employed to mitigate notoriously known problem of CBSE system — performance overhead (caused e.g. by intercomponent communication). Finally, functional components and the runtime platform are assembled together to form the resulting application, Fig. 1a step 3.

We distinguish two types of development roles involved in this process — application developer and framework developer. Application developer is responsible for development of functional components and specification of domain-specific requirements — in Fig. 1a step 1. The role of the framework developer is to design and implement the runtime platform generation process, and the domain-specific requirements defined by the application developer — in Fig. 1a step 2 and 3.

Our Proposal. Considering the presented process, we can notice that for each domain, a different process is used. However, the steps 2 and 3 share many similar concepts across different application domains (from code generation tasks, application instating and deployment, to tool-chain development etc.). Moreover, they are usually implemented in an ad-hoc manner without any reuse. We therefore propose a new development process presented in Fig. 1b. As the cornerstone we use a generic component model that is easily extendable towards different application domains, in Fig. 1b step 1. Consequently, since all domain specific models share the same concept, an unified approach to runtime platform generation can be employed in steps 2 and 3. Therefore, we can summarize the key contributions of our paper:

• A Generic Component Model and Domain Components. We propose Domain Components — a unified approach to specification of domain-specific requirements presented in custom containers. This allows the application developer to easily manipulate domain specific requirements since they are represented as first-class entities and are separated from the functional concerns. Furthermore, we identify common patterns that are used by framework developers to implement semantics of Domain Components.

• A Framework To Build Component Frameworks. We develop a framework, in the literature also refereed as meta-framework [5], composed of high-level tools, methods, and patterns allowing framework developers to generate runtime platforms in a generic way according to concerns captured by Domain Components. Within our approach, the platform is built using component assemblies and is based on our generic component model. Moreover, since we are able to reason about the whole system (functional and platform concerns) using common concepts (components, assemblies), various architecture optimizations independent from the target domains can be introduced, which contributes to better performance of resulting applications.

III. CONSTRUCTING DOMAIN-SPECIFIC COMPONENT FRAMEWORKS

In this section, we introduce the basic concepts of our generic framework presented in Fig. 1b. As the cornerstone of our proposal we define a generic component model, depicted in Fig. 2. The model is divided into core- and platform-level. First, in Sect. III-A, we present the core-level concepts and introduce Domain Components — special components for expression of domain-specific concerns in the application. Furthermore, responsibilities of application and framework developers are exactly defined. Second, in Sect. III-B we introduce the runtime platform construction process — the architecture refinement. In this process, framework developer refines the application architecture through the architectural patterns that we define in the platform-level of the model.

A. A Generic Component Model

The model is based on the popular CBSE principles [4], containing the basic entities Component, Interface, Binding, Primitive and Composite component. Moreover, the component model adopts the sharing paradigm [4] — one specific component can be a sub-component of more than one composite component.

A brand new entity that we introduce is Domain Component, inspired by [21]. The main purpose of domain components is to model domain-specific requirements in a unified way. Within our model, domain components are reified as composite components. The sharing paradigm allows developers to fully exploit this concept. By deploying subcomponents into a certain domain component, the developer specifies that these subcomponents or the bindings between them support the domain-specific property represented by the domain component. Moreover, a domain component contains a set of attributes parameterizing its semantics.
The approach of modeling domain-specific aspects as components brings advantages commonly known in the component-based engineering world such as reusability, traceability of selected decisions or documentability of solution itself. Also, by preserving a notion of a component, it is possible to reuse already invented methods (e.g. model verification) and tools (e.g. graphical modeling tools) which were originally focused on functional components. If we go further and retain domain components at runtime then it is possible to reconfigure non-functional properties represented by domain components on-the-fly.

We illustrate the DomainComponent concept in Fig. 3a. Components Writer, Readers, MailBox, Library and their bindings represent a business logic of the application. The domain component DC1 encapsulates MailBox and Library, thus defining a domain-specific service (e.g. logging of every interface method invocation) provided by these two components. At the same time, component DC2 represents a different service (e.g. runtime reconfiguration) and defines that this service will be supported by components Writer and Readers. Therefore, the domain-specific concerns are now represented as first-class entities and can be manipulated at all stages of component-software development lifecycle.

The role of the application developer is therefore to create and implement functional components and to specify domain-specific requirements by deploying functional components into domain components. While the application developer is aware of the semantics behind domain components, he does not provide their implementation and therefore can fully focus on functional concerns of the application. To give an example, a domain-specific component ThreadArea can specify execution context (an executing thread and its properties) of an active functional component, however, the application developer does not need to know how these properties are enforced at runtime. For more examples see Sect. V. Moreover, it should be highlighted that the semantics of domain components could impose constraints over the functional components’ implementations (e.g. a component within a ThreadArea should not instantiate new threads). The expression and the checks related to such constraints is out of the scope of this paper and is addressed in [19].

The role of the framework developer is to define and implement semantics of domain components. First, his responsibility is to define domain components according to the needs of application developers and to define the rules constraining application of domain components at the functional level. Afterwards, the framework developer designs and implements semantics of domain components using the platform-level concepts — see Fig. 2, and the architectural patterns that we introduce in Sect. III-B.

### B. Architecture Refinement of Domain Components

The key role of the framework developer is therefore to implement semantics of domain-specific components. When considering domain components and the functionality they express, they impact two core architectural concepts: Functional Component and Binding. We further refer to this phenomenon as architectural refinement of core-level concepts through the platform-level concepts.

A functional component typically implements the business part — a code provided by the application developer, and requires the platform part that implements the domain-specific services — the container. By the Functional Component Refinement we mean that the set of domain specific services is determined by the domain components, consequently the container architecture of a functional component is refined with according platform concepts. A domain-specific service can also pose special requirements on the intercomponent communication (e.g. logging, broadcast communication management), in these cases we speak about Binding Refinement. From our current experiment in using our approach, we claim that these two refinement points (components and bindings) allow to specialize the core-level concepts according to arbitrary domain-specific requirements. We therefore define two architectural patterns, in Sect. III-B1, that address the challenge of the architectural refinement and allow framework developers to develop properly implementations of domain-specific concepts.

1) Architectural Patterns: The key purpose of architectural patterns is to allow framework developers to define semantics of domain components and thus to refine the application architecture in a systematic and programmatic way. The patterns are designed to implement any type of a domain-specific service that can potentially be reflected by a container, they therefore define architecture invariants, design and composition rules for the platform-level. In Fig. 2, the platform-level presents two architectural patterns: ChainComposite and ContainerComposite, and we clarify them in the remainder of this section.

*ChainComposite Pattern* is defined as a composite component, the subcomponents of such a composite are special components — *interceptors*. Within the ChainComposite pattern, the interceptor components are bounded via their incoming and outgoing interfaces in an acyclic list, as depicted in Fig. 3a. Here, the IN and OUT interface signatures of the interceptors are not necessarily identical, this allows developers to identify interceptors as *adaptors* of the intercepted execution flow.
The interceptor itself could be a composite component allowing framework developer to implement complex intercepting mechanisms. The ChainComposite component at the platform level refines a binding specified at the functional level, thus the pattern is similar to the concept of the connector [17].

ContainerComposite Pattern, initially introduced in [20], is also specified as a composite component and reifies a container of a functional component. As defined in Fig. 2, it is composed of ChainComposite components and Controller components. The ContainerComposite pattern is applied on a primitive (see example in Fig. 3b) or composite functional component as follows:

- A set of Controller components implementing various domain-specific services and meta-data influencing the whole component (e.g. lifecycle management, reconfiguration management) is composed in the container. Moreover, a special control interfaces are provided to allow an access to these services from outside of the component.
- For each interface of the functional component a ChainComposite pattern is used. ChainComposite components can be interconnected by TRAP interfaces with the controllers, thus allowing centralized management of strategies for interception mechanisms.

2) Architectural Refinement Process: Once we specify the functional architecture containing domain components and also architectural patterns for these domain components we employ the architecture refinement process — a process where the core-level architecture specified by the application developer is refined into an architecture where both functional architecture and runtime platform architecture are designed using the platform-level concepts. As a result of this process we obtain a runtime platform architecture where both functional and domain-specific concerns are represented. The crucial point of the architecture refinement process is therefore the propagation of domain-specific concerns into the architecture. The important feature of the architecture refinement process is its variability and extensibility to allow employing different refinement strategies as well as support for new domain-specific components, validation and optimizations. All properties stated above are reflected in the implementation of the architecture refinement process called HULOTTE, described in Sect. IV.

IV. HULOTTE FRAMEWORK

In this section we describe HULOTTE framework — an extensible tool-set that we have developed to implement the architecture refinement process. However, rather than to implement the whole process in a single transformation step that can be error-prone and hard to extend, we employ a step-wise refinement process [2] in order to refine the high-level concepts in our architecture gradually in several stages. This technology allows framework developers to easily modify and extend this process with new domain-component definitions and semantics. Consequently, we employ methods of generative programming to compose functional code implemented by the application developer with the runtime platform implementation.

To develop the framework, we have applied the technology for development of extensible tool-sets introduced in [15]. HULOTTE is thus developed purely using CBSE paradigm allowing framework developers seamless extensions towards different refinement strategies. The HULOTTE framework, depicted in Fig. 4, consists of three main units — front-end processing a description of a functional architecture stored in ADL, middle-end responsible for a step-by-step architecture refinement, and backend which serves as a target domain specific implementation generator.

Front-end implements the translation layer. It allows us to process architectures represented by different notations (e.g. Fractal-ADL [21], UML, ACME) and to transform them into an independent internal EMF-model based representation. The translation process can be extended by appending a new translator component. The new translator typically reflects a domain-specific extension of ADL (e.g. DistributedNode, ThreadArea presented in Sect. V).

Middle-end is the central part of the HULOTTE framework and implements the refinement process. Its task is to process the architecture description in the form of the EMF model produced by the front-end, apply defined architecture refinements — creating, connecting, or merging model elements according to employed transformations. Internally, the middle-end is composed of three processing
units — PlatformBuilder, Validator, and Optimizer.

PlatformBuilder is responsible for the model refinement and consists of a chain of component builders (for implementations of interceptors, controllers, and components) where each chain participates in the refinement process. From the builders the runtime platform components are instantiated either by loading definitions from an off-the-shelf component library or programmatically, via the high-level API provided by the framework. The selection and execution order of chains is controlled by MainBuilder Dispatcher that recursively explores the platform architecture and applies appropriate builder chains. Moreover, refining the internal structure as a chain of ComponentBuilders encourages extensibility of the whole process, since a new domain-specific builder can be easily introduced.

Validator verifies that resulting platform architectures are in conformance to the architectural constraints and invariants of domain components. The task is not only to verify whether the architectural patterns were applied correctly but also to assert that domain components were specified with respect to their constraints (e.g. to arbitrarily apply two different domain components over the same functional component is sometimes not meaningful, see the Limitations of the Approach in Sect. V-C). The validator part of Hulotte is detailed in [19].

Optimizer introduces optimization heuristics in order to mitigate the common overhead of component-based applications. The heuristics focus on reducing interceptions in inter-component communication which usually causes performance overhead, and on merging architecture elements in order to decrease memory footprint. A detailed description of the heuristics provided by our framework is out of the scope of this paper, we refer the interested reader to [21]. Moreover, since a complete architecture of the system is available at this stage, additional architecture optimizations can be introduced while still being independent from the target domain.

Back-end permits generation of different types of target implementations according to deployment requirements (e.g. C for embedded devices, Java for enterprise applications). The back-end part of the framework is also highly configurable in order to reflect current target domain and chosen implementation language. Currently, the back-end is a collection of Java code generators generating Java classes from particular model elements.

V. EVALUATION

A. Case Study: A Framework for Real-time Java based Systems

The initial case study introduces a component-based framework for RTSJ-based real-time and embedded systems [21]. As the cornerstone of the framework we have defined a domain-specific component model [23] which fully reflects the specifics of Real-time Specification for Java (RTSJ) [3]. The key motivation for this case study is to employ the domain component concept and the HULOTTE framework in order to achieve a better separation of concerns of RTSJ systems and to mitigate complexities of the RTSJ-based development process.

![Figure 5. Real-Time Collision Detector](image)

Therefore, the domain component concept is used to represent RTSJ concerns. We define MemoryArea domain component to express different allocation areas of RTSJ systems — heap, scoped memory, and immortal memory. Furthermore, ThreadDomain component is defined to represent various execution concepts enforced by RTSJ — non-realtime, real-time and non-heap real-time, and to distinguish between active and passive functional components. Consequently, the HULOTTE patterns were used to implement defined domain components. The ChainComposite pattern is employed to implement MemoryArea components by providing correct switching between allocation contexts and supporting cross-area communication. Similarly, the ContainerComposite patterns implements containers of components deployed in the ThreadDomain component.

Real-time Collision Detector. To apply our domain specific framework, we have implemented a large case study — Real-time Collision Detector (RCD) introduced in [1]. The RCD algorithm is about 25K Loc and its task is to proceed a periodic stream of aircraft positions and determine if any of these aircrafts are on a collision course.

Figure 5a shows a snippet of the RCD architecture designed in our approach. The Runner component represents the starting point of the application, by deploying
it in the ThreadDomain:NHRT component we precisely define its execution context.ScopedMemory2 encapsulates functional components responsible for computations performed in every iteration of the algorithm and implements deallocation of temporal data between every iteration, results of these computations are stored in a StateTable component defined as a persistent by the ScopedMemory1 domain component. Finally, the patterns introduced in Sect. III-B were employed to implement domain components. In Fig. 5b we demonstrate application of the ChainComposite pattern that implements cross-scope communication between MotionCreator and StateTable component; and application of the ContainerComposite pattern that was used to implement the ThreadDomain for the Runner component.

Evaluation. When developing the RCD example we can witness several benefits of our approach. The domain specific component model [23], designed through the domain component concept, allowed us to construct a specific framework [21] addressing fully the challenges of RTSJ-based software development. The domain components simplified expression of RTSJ specific properties, since these properties are present in the architecture as first-class entities. A full separation of functional and real-time concerns is achieved, therefore, the functional code is more readable — reflecting the functional needs of the application without any constrains imposed by the real-time properties. As the second benefit of our approach we consider application of the HULOTTE toolchain for automatic generation of the runtime platform implementing RTSJ-related code, which is highly error-prone when implementing by hand. Moreover, performed benchmarks published in [21] showed that our approach does not introduce any overhead comparing to purely object-oriented methods.

B. Other Applications and Lessons Learned

Apart from the presented case study, we have validated our approach in studies spanning various domains.

In [16] we have introduced a new domain component — DistributionNode (DN), to address challenges of distribute applications. A functional component in DN will be thus deployed on the corresponding node together with its runtime support extended towards the specifics of distributed communication. The role of the framework developer is therefore to apply the ChainComposite pattern on each distributed binding, Consequently corresponding stubs and skeletons will be refined as a subcomponents of the ChainComposite and automatically generated by the HULOTTE framework. Moreover, the framework generates each DN component as a self-standing application allowing deployment of the components into the corresponding nodes. The evaluation showed that the approach brings significant ease into a distributed system development.

In [22] we have addressed the challenges of ambient and ubiquitous computing by designing dedicated domain components. The HULOTTE framework allowed us to implement specifics of ambient communications using the architectural patterns.

Furthermore, we have conducted a study on reflective and reconfigurable services, a challenging topic addressed by many component frameworks [4], [6], [9], [13]. In our approach, such requirements can be specified by dedicated domain component - ReconfigurationDomain. The achieved result corresponds to the Fractal component model [4]. The non-functional properties are specified by domain components and architectural patterns are used during their development, see Fig. 6, thus reducing the complexity for application developers. Moreover, our approach outperforms Fractal by using the Domain Component concept, since we are able to exactly specify where the introspection and reconfiguration services will be supported, which allows us to pay for the runtime flexibility only where needed.

When conducting these studies, we learned that the approach scales very well for various domain-specific properties orthogonal to business logic, while reducing complexity of developed applications. Moreover, the framework structure allowed us to easily extend the set of supported domains.

C. Limitations of the Approach

In this paper we focus on definition of domain components and their integration in the HULOTTE framework. However, an open research issue still remains specification of policies and constraints that regulate application of domain components at the functional level. Since some domain-specific services are non-orthogonal - competing or dependent on each other, their application must be exactly delimited in a form of policies that will manage non-trivial combinations of domain-specific services. This is however out of the scope of the paper, we plan to pursue this topic in our future work.

VI. RELATED WORK

Applying generative methods [11] to propose a general approach to component framework development is not a novel idea. Bures et. al. [5] summarize properties and requirements of current component-based frameworks and proposes a generative method for generating runtime platforms and support tools (e.g. deployment tool, editors, monitoring tools) according to specified features reflecting demands of a target platform and a selected component model. Comparing to our approach, the authors provide the
similar idea of generation runtime platform, however they merely focus on runtime environment and related tools and neglect a definition of component model requirements by claiming that the proposed approach is generative enough to be tailored to reflect properties of contemporary component models.

Similarly, Coulson et. al. [9] argue for the benefits and feasibility of a generic yet tailorable approach to component-based systems-building that offers a uniform programming model that is applicable in a wide range of systems-oriented target domains and deployment environments.

Furthermore, many state-of-the-art domain-specific component frameworks propose a concept of containers with controllers refined as components, e.g. DiSCo framework [24] addressing future space missions where key challenges are hard real-time, embedded constraints, different levels of application criticalities, and distributed computing. Cecchticky et al. [7] presented the generative approach to automating the instantiation process of a component-based framework for such on-board systems.

On the other hand, aspect-oriented programming (AOP) is a popular choice to non-functional services implementation. Moreno [18] argued that non-functional services should be placed in the container and showed how generative programming technique, using AOP, can be used to generate custom containers by composing different non-functional features. This corresponds with our approach, however, as we have shown in [20], aspects can also be represented as domain components — AspectDomain component, thus allowing developers to leverage the aspect-techniques to the application design layer, and to represent them as components.

VII. Conclusion and Future Work

The recent boom of domain-specific component frameworks (DSCF) brings a challenge of constructing them effectively by enforcing reuse, since ad-hoc approach to their implementation still dominates.

This paper brings the following contributions. First, we have proposed Domain Components — a unified approach that clarifies specification and non-functional capabilities of domain-specific requirements presented in custom containers. Moreover, we have identified common patterns that facilitate implementation of Domain Component semantics. Second, we have developed a generic framework that uses generative programming methods to instantiate domain-specific applications together with their runtime platform. Finally, the whole approach is highly transparent since it is based on a component model with a configurable tool-set, which allows developers to easily extend it towards various domains.

To evaluate benefits of our approach, we have conducted various case studies that span different domain challenges. The results showed that our approach supports clear separation of functional and non-functional concerns of applications. Furthermore, proposed architectural patterns together with employed generative programming methods mitigate complexities of implementation of domain-specific concerns. Additionally, as we have shown in [21], our approach introduce various optimizations that reduce the usual overhead of component-based applications.

As for our future work, an open research issue still remains consistent and symmetric approach to construction of containers needs to be specified in a form of policies that will manage non-trivial combinations of domain-specific services.

ACKNOWLEDGMENT

The Collision Detector [1] was originally developed by Filip Pizlo et. al. at Purdue University, we would like to thank Prof. Jan Vitek for providing us the source codes. This work was supported by the Czech Science Foundation under the contract no. 201/09/H057, and by the french ANR/RNTL FLEX-EWARE project.

REFERENCES


