Automatic Synthesis and Adaption of Gray-box Components for Embedded Systems — Reuse vs. Optimization

Etienne Borde
Institut TELECOM, TELECOM ParisTech, LTCI
Paris, F-75634 CEDEX 13, France
Email: etienne.borde@telecom-paristech.fr

Jan Carlson
Mälardalen Real-Time Research Centre,
Mälardalen University, P.O. Box 883,
SE-721 23, Västerås, Sweden
Email: jan.carlson@mdh.se

***Abstract*—Component-based development of embedded systems has been suggested as a means to increase development efficiency by, for example, facilitating reuse. However, the specifics of the embedded systems domain also raise some particular difficulties when applying this approach. For example, when glue code is automatically produced from an architectural specification, a systematic approach where fully reusable code is generated for all entities in the system, can lead to unaffordable overhead in embedded systems with severe resource limitations and temporal constraints. If, on the other hand, highly optimized code is produced by taking advantage of the specific context in which each component is used, then the generated code is not reusable in other contexts, and the potential benefits of component-based development are not fully exploited.**

In this paper, we present a component-based framework that permits a more detailed trade-off between optimization and reusability, by automating the integration of components for which the software designer can specify the desired reuse potential. Depending on this specification, the integration code is either reused and adapted, or completely optimized.

***Keywords*-integration; code generation; component-based software engineering; reuse; optimizations;**

I. Introduction

More and more products in our everyday life include functionalities that are provided by embedded software systems. In order to cope with an increasingly competitive market, these functionalities are more and more sophisticated, and thus increasingly complex. For the same reason, the design of such embedded systems must cope with a constant need to shorten time-to-market.

In order to accelerate the design of such systems, component-based software engineering (CBSE) proposes to build them by assembling well identified subsets of the software functionalities, called software components. One of the benefits of such an approach is the possibility to reuse and integrate existing components, possibly developed externally. In this context, software components are most of the time of *gray-box* nature: even if their behaviour can be captured by the component model semantics or some external specifications, their internal implementation is not exhaustively modelled.

Although gray-box component models have been successfully used in general purpose software engineering, their adoption in the domain of real-time embedded systems still raises important challenges [1]. In particular, these systems often have heterogeneous non-functional constraints, which makes the reuse of existing code in different usage contexts difficult. Reuse of code has a greater potential impact on development time than reuse only at model-level, since it allows for reuse of code-level analysis and test results together with the code.

Indeed, generating code for a fully reusable entity requires considering that this entity might be reused in any usage context, and the overhead of the resulting code may be unaffordable due to the required general mechanisms. On the other hand, synthesizing code for one particular usage context leads to produce optimized entities although loosing the potential benefits of a CBSE approach: component independence and reusability.

In this paper, we present a component-based framework that automates the integration of reusable gray-box components. This framework helps in realizing the trade-off between reusability and optimization by proposing:

i) an architecture of the generated code that separates clearly three different levels of reusability and optimisations,
ii) a general purpose implementation of the generated code that aims for reusability of the produced entity; and
iii) optimization algorithms that address the fusion of hierarchically nested components.

The remainder of this paper is organized as follows: In Section II we present ProCom, the component model our approach is based on. Sections III, IV, and V present respectively the architecture of the generated code, the principles of its implementation from a reusability perspective, and the adaptation of an implementation to a particular usage context. Finally, we compare our contribution to related works in Section VI, before Section VII concludes the paper and outlines future work.

II. Background – ProCom, the Component Model

The proposed synthesis approach has been investigated in the context of ProCom [2], a component model specifically targeting the domain of distributed real-time systems. In this section, we present those aspects of the component model that have a significant impact on the synthesis process.

To address the different concerns that exist on different levels of the design of such systems, ProCom consists of two distinct, but related, layers. The upper layer models a distributed embedded system as a number of active and concurrent subsystems, communicating by message passing. Our synthesis approach, however, is concerned with the lower layer which addresses the internal design of a subsystem. That layer is based on a notion of passive components, and the communication between them follows a pipes-and-filters architectural style with an explicit separation between data and control flow. The former is captured by *data ports* where data of a given type can be written or read, and the latter by *trigger ports* that control the activation of components.

In order to implement complex functionalities, components can be connected by simple *connections* that transfer data or control, or *connectors* providing more elaborate manipulation of the data- and control flow.

ProCom is hierarchical, meaning that components can be internally constructed by a set of interconnected subcomponents, possibly in several levels of nesting. Contrasting such *composite* components, the *primitive* components at the bottom of the hierarchy are implemented as C functions.

Figure 1 shows the model of a composite ProSave component. Trigger and data ports are denoted by triangles and rectangles, respectively, and the filled circle is a compact representation of *data fork* and *control fork* connectors.

A main characteristic of ProCom is that the control flow is very explicitly captured in the architectural model. This is partly due to the separation of data- and trigger ports, which allows the control flow to be modelled by trigger port connections, but more importantly it is a consequence of the severe restrictions imposed on the component behavior by the component model.

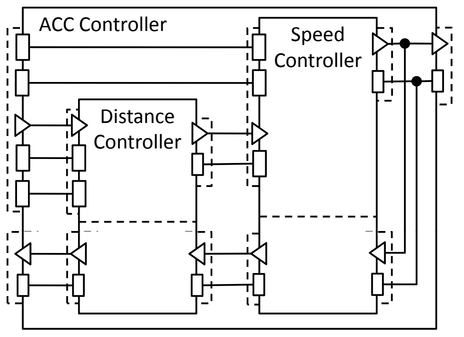


Figure 1: Composite ProCom component

Ports are structured into *port groups* (denoted by dashed boxes in the figure) consisting of one trigger port and a number of data ports. The data of a group are always produced or consumed together, in an atomic and conceptually instantaneous action, when the trigger port of the group is activated.

The functionality of a component is provided as a set of *services*, each consisting of one input port group and a number of output groups. In Figure 1, the composite component consists of a single service with two output groups, and the two subcomponents have two services each, denoted by the dashed lines.

Services are triggered individually and can execute concurrently, while sharing only internal state data. The data at the input port group are accessed at the very start of each invocation, and any subsequent writing of data to the input ports will not affect the component until the next invocation. When each output port group has been activated once, the service changes from *active* to *idle* state.

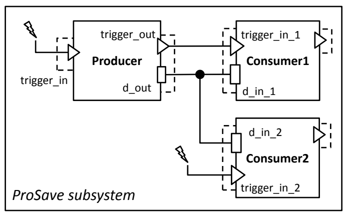
These restrictions serve for tight read-execute-write behavior of a service, but they also mean that the control flow can be determined without knowledge of the component's internals. Another restriction, stating that an activation of a service that is already active is simply ignored, avoids the problem of multiple concurrent, and possibly overlapping, activations of a service.

III. Architecture of the Generated Code

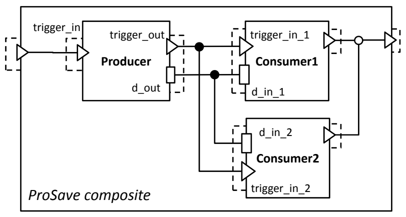
The architecture of the generated code aims at implementing a trade-off between reusability of existing code and adaptation to a specific usage context. To achieve this objective, we have identified possible variations in the implementation of the ProCom semantic. These variations depend on the architectural context in which a component is used as a subcomponent. As an example, Figure 2 shows three different usage contexts of the components *Producer*, *Consumer1* and *Consumer2*. The impact of this variation on the implementation of the component wrappers will be addressed in Sections IV and V.

We present in this section how the architecture of the generated code has been designed in order to deal with this variability while reusing as much as possible the code that has already been produced, and possibly validated. The basic principle of this architecture is to externalize all variable parts in a *service handler* implementation, while the interfaces definition and the internal implementation represent the reusable view of the service implementation.

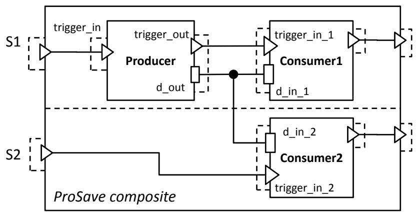
Figure 3 represents the data structures generated for a composite component. This architecture is divided in three main parts. The first one represents the component *interfaces*, a data structure that can be reused in any usage context of the component. This architectural layer consists of two data structures: The *Component* data structure represents a



(a) Dispatched by external events



(b) Dispatched in a single service



(c) Dispatched in two services

Figure 2: Different usage contexts of the same subcomponents

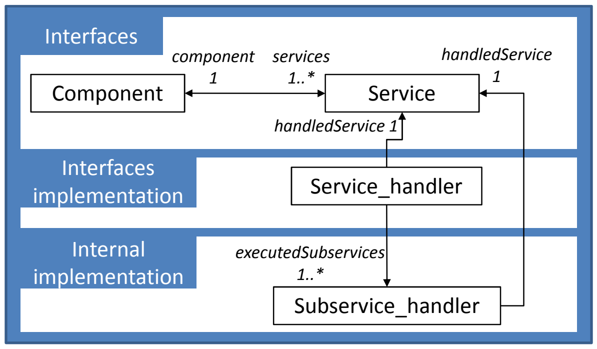


Figure 3: Generated Code Architecture

component instance, and also defines a reference to the internal state of the component, to be used by the functional code to define, initialize and access the internal state. The *Service* data structure corresponds to a service of this component, including references to its input and output ports.

The second part represents the *implementation* of those interfaces, and corresponds to the external, variable part of the generated code in that it be adapted to a specific context. The *Service_handler* data structure is the main constituent of this layer. This entity represents a context dependent implementation of the service interfaces, and will be replaced when the user context is known. It implements context-independent trigger operations (in and out) as well as data transfer operations (in and out).

Finally, the third layer represents the *internal implementation* of a component. This layer implements the behaviour specified by connections to (and between) subcomponents inside the composite, and remains unchanged independently of changes in the usage context of the component. In some situations it might be beneficial to allow adjustments also in this layer, but such an extension of the approach is not addressed in this paper. The *Subservice_handler* data structure corresponds to the internal implementation of a composite component. This entity gathers the definition of subservices, either in reusable form or a re-synthesized version.

This architecture separates the variable and stable parts of the generated code. Based on this architecture, we present in the next section the synthesis of component code that can be used in any context. After that, in Section V, we describe how components are adapted for a given context.

IV. General Purpose Synthesis

When a component is independently synthesized, the synthesis tool does not have any information about the context in which the component will be used. As a consequence, the production of a reusable unit requires implementing the component interfaces in the most general case, while ensuring the respect of the component semantics.

A. General Interface Implementation

Some aspects of the ProCom semantics were presented in Section II, and we extract from this specification the following properties that have to be ensured for the generated code to satisfy the semantics independently of context:

1) **Stable data**: Input ports are read at service invocation, and the resulting values must remain unchanged throughout the computation, independently of new data arriving to the ports.
2) **Service locking**: Activations arriving to an already active service are ignored.
3) **Atomic group data transfer**: The data of the output data ports in a group are all transferred to the connected components in a single atomic action, performed when the trigger port of the group is activated.

To tackle the issues of stable data and atomicity, we propose a double buffer implementation for data interactions: the output port of a component is represented by two buffers, one that can be updated during the execution of the component, and one that contains the last up-to-date value. The transfer of data from producer to consumers then consists of just sending a reference to the up-to-date buffer.

When a service is triggered (i.e., when the trigger port of the input port group is activated), service locking is implemented by first checking if the service is already active. If in idle state, data for each input port group is copied to an internal data structure from the source of the last reference received. This internal structure remains unmodified during the execution of the service.

In order to ensure data atomicity and consistency, this copying of data into a consumer must never be performed simultaneously with (or interleaved by) the action transfering data from the producer. This is ensured by executing the trigger operation in a dedicated critical section, as further detailed in [3]. The double buffer solution, described above, is preferred over a single buffer implementation, since it reduces time spent in the critical section (only a pointer switch on the writer side).

Listing 1 illustrates service locking and data transfer operations as they are generated for the service handler of component *Consumer1* from Figure 2(a), independently of any usage context.

```
int Consumer1_t_in1 (Consumer1_svc * svc){
  // Ignore trigger if already active
  if(svc->active) return 0;
  else {
    svc->active=1;
    Consumer1_transfer_d_in1(svc);
    return 1; // Schedule the component's execution
  }
}

void Consumer1_transfer_d_in1(Consumer1_svc * svc) {
  // Copy external->internal view (cnx_d_in1->d_in1)
  svc->d_in1=svc_h->cnx_d_in1;
}
```

Listing 1: Component interface implementation

B. Internal Implementation

One of the main advantage of the ProCom component model is the ability to explicitly model the control and data flow. Thus, in most cases the internal behaviour can be achieved by a hardcoded sequence of calls invoking subcomponent services.

However, there are constructs in ProCom that allows the definition of systems where the control-flow is only partially known statically. One such construct is the *selection* connector, by which data dependent control flow decisions can be defined. Another case is when there are subcomponent services with multiple output groups, and that subcomponent is reused rather than re-syntesized in this particular context. The semantics states that for each activation of the subservice, each of the output port groups will be triggered exactly once, but the order in which ports are triggered is not known at design time.

Thus, in the general case, the generated code must enable the orchestration of components based on information received at runtime. To handle this, the enclosing component stores information about activated subservices in a list and schedules the corresponding subcomponents according to a predefined orchestration policy.

V. Reuse and Adaptation

Whenever a component is used (or reused) within a larger system, the external part of the component implementation (i.e., the interface implementation) can be adjusted based on information about that particular context, to reduce the overhead.

Generally speaking, the usage context of a component instance consists of all entities in the enclosing unit that is being synthesised, and attributes associated with them or with the system as such, including for example temporal requirements and the mapping of components onto the execution platform. It also includes attributes associated specifically with this instance of the reused component. The detailed adaptation algorithm proposed here is mainly based on architectural information, and extending it to pay more attention to non-functional requirements will be addressed as future work.

As an example, we consider the three contexts illustrated in Figure 2. In (a) and (c), the writing of data to port d_in_2 of the Consumer2 component is potentially performed in a different thread than (and thus possibly concurrently with) the triggering and execution of the component. Thus, in these cases, the general interface implementation, with double buffers and locking, is needed to ensure data consistency and conformance to the ProCom operational semantics. However, in (b), since the producer and consumer belong to the same service in the enclosing composite, and since the triggering order between them can be determined statically, the interface implementation can be optimized to use a single buffer and no locking, without modifying the internal implementation of the component. The details of this adaption are described later in this section.

A key objective of our approach is to take advantage of the usage context information in order to reduce the overhead in the general purpose service handlers and the generated glue code. To accomplish this, we propose a set of model transformation steps in order to achieve an optimal fusion of components and subcomponents in that particular context. The main requirement of this fusion process is to preserve the integrity of the components semantic.

Given a composite component to synthesise, the role of our synthesis process is to produce an optimized reusable binary code library for that component based on the architecture defined in Section III.

The main steps of the synthesis process are the following:

1) Transform the hierarchical model into a single flat model (down to components that are primitive or reused).
2) Extract the control and data flow information.
3) Refine the order of service execution.
4) Generate optimized interface implementations for subcomponents.
5) Generate the interfaces and general interface implementation code for the synthesised entity, and code realizing the internal control- and data flow.

In the remainder of this section we present in more details these different steps.

1) Model Flattening: The model flattening is completely safe with regards to preservation of the operational component semantics. This step consists of representing the whole tree of nested subcomponents as a single collection of connected primitive and reused components. Instances of composite components are removed, and their internal subcomponent structure is added to the enclosing entity, together with constructs that emulate the semantics of the removed composites where needed.

2) Control and Data Flow Extraction: From the result of step 1, we build graphs corresponding to the control- and data flow of the synthesised entity. For each service of the synthesised entity, there is an associated set of subservices (i.e., services of subcomponents) and a set of *progress steps* representing the different advancement points in terms of control flow, such as calling the entry function of a subservice, evaluating the guard of a selection connector, etc.

3) Control Flow Refinement: Without any requirements on the synthesised component, it is not possible to make wise choices about the optimal ordering of multiple interactions (as presented in section IV). In our future work, we plan to integrate non-functional requirements such as timing and memory footprint requirements to lead this decision. For now, the synthesis tool implements the straightforward choice to execute forked paths in sequence, and to finish the control path leading out of one port group before returning to resume the subcomponent.

4) Adapting Subcomponent Interface Implementations: As described previously, the general interface implementations of the subcomponents can be adapted to their particular context in this synthesised entity. Concretely, two main optimizations are applied to simplify the locking and double buffer protection against concurrent reading and writing of data, where applicable.

First, concurrent reading and writing of a data port is not possible if all writing and reading of the port are performed in the same enclosing service of the synthesised entity, since the semantics prevents multiple threads from executing the service concurrently. Thus, in these situations the costly locking mechanism is not required, and can be removed.

Second, the role of the double buffers is to ensure that data transfer takes place only when the trigger port of the group is activated. The condition under which this holds trivially without an additional buffer for the data port p is that for all ports p' receiving data from p, the following holds:

i) p and p' belong to subcomponents, not to the synthesised entity; and

ii) The port group of p is either always triggered before the port group of p', or always triggered after the full execution of the subservice that p' belongs to.

Removing the double buffering means that the reader and writer can communicate by a single shared variable, without breaking the strong limitations in the component model restricting communication to occur only when a port group is triggered.

5) Code Generation: As the final step of the synthesis process, code is generated for the synthesised unit. Following the architecture described in Section III, the generated code consists of interfaces, a general interface implementation, and an internal implementation based on the control-flow representation and references to the subcomponents (primitive or reused).

VI. Related Work

Many component frameworks already exist both in industry and in academic research. The focus of this this section is approaches that support gray-box components, i.e., their internal behaviour is not exhaustively known, and address synthesis and integration of reusable components in embedded systems.

CAmkES [4] is a component model dedicated to the design of real-time operating system focusing on security properties. The interaction semantic of this component model is based on three paradigms: interface based, event based, and data based. Data interactions are meant for representing shared memory space between two software components. As a consequence, this component framework requires identifying, during the design, the existence of shared memory accesses.

CIAO [5] and MyCCM-HI [6] is a component framework dedicated to the adaptation of the Lw-CCM[1] standard to the domain of real-time and embedded systems. The scope of the work presented in [5] is very close to the scope of the contribution presented in this paper insofar as it aims at automating the fusion of components in order to meet the strict non-functional requirements of embedded systems. To that respect, the component-based architecture presented in this paper defines a specific structure (namely *context*) that has to deal with modifications of the component's usage context. However, this approach relies on a semantic, initialization process and underlying middleware that limits its usability in very constrained (in terms of memory, performances, and predictability) embedded systems. Indeed, the architecture focuses on flexibility of the design and thus relies on dynamic initialization and configuration of the components' data structure. In [6], the authors use a static deployment and configuration approach on top of a middleware dedicated to embedded systems. However, this work focuses on adaptive systems, not on the reuse/optimization trade-off.

Contrasting with our approach, the component models used in these frameworks does not impose any restriction on the component's behaviour: once a component is activated, its output interfaces *may* (not *must*) be activated. As a consequence, our framework automates the decision of the usage

[1]Standard from the Object Management Group: Light Weight CORBA Component Model Revised Submission; Document realtime/03-05-05 edn.

of shared memory by *i)* using a more abstract specification of the interactions and *ii)* taking advantage of the semantic restriction imposing that when a component is activated, its output interfaces *must be triggered once and only once*.

Component adaptation is also addressed by, e.g., Canal et al. [7]. The main difference compared to our work is that they use interface adaptation as a means to compose components that are not fully compatible, while our goal is to increase efficiency without violating a restricted communication semantics.

In the scope of model driven optimizations for embedded systems, two works are of particular interest. Both use AADL[2], a component-based language in which the modelling artefacts represent concrete entities of the software and hardware architecture (processes, data, subprograms, processors, memory, etc.). Besides, AADL defines a precise execution semantic for each of the software components that enables the formal analysis of the whole system.

Based on the Ocarina tool suite, that aims at synthesizing AADL model into different programming languages, [8] proposes an approach to fusion the different activities (or threads) of a real-time embedded system. This work is a complement to the contribution presented in this paper, and the proposed fusion of activities can be an interesting result to be used as an input of our optimizations.

In ArcheOpterix [9], the authors propose some heuristics to automate the decision of the software-to-hardware allocation, thus improving different quality attributes of the architecture. The results presented in this work are also complements of our contribution.

Besides complementary, our contribution pursues a different objective: reusability of generated code. Both [8] and [9] focus on optimizations at a different level of abstraction, closer to the final realization of the overall system.

VII. Conclusion and Future Works

The integration of software components is a difficult task, especially when addressed in the domain of embedded systems. Indeed, it requires implementing the interactions between those components not only in order to provide the desired functionalities but also to ensure the respect of non-functional properties. Considering the effort required to reach this objective, a satisfactory result should be reused in future evolutions of the system, or even in other systems requiring the same functionality.

In general, component-based software engineering has been a fruitful solution for both integration and reuse of existing software. However, its adoption in the scope of embedded systems still raises important challenges. In particular, we have shown in this paper that the usage of CBSE in embedded systems implies a trade-off between reusability and optimization of the corresponding component. To tackle this issue, the principles presented in this paper help in synthesizing code for components integration. Besides, we propose in this paper a set of optimization steps that aim at reducing the overhead (in terms of memory footprint and execution time) due to the presence of glue code.

As future work, we need to extend our approach by taking into account more information about the usage context of a component, like non-functional properties for instance. This could help the software designer in deciding the level of optimization/reuse of a component. The treatment of non-functional properties to decide on the ordering of components is also part of our future work.

[2]Standard from the Society of Automotive Industry: Architecture Analysis and Design Language

Acknowledgment

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre Progress, and by the Swedish Research Council project Contesse (2010-4276).

References

[1] I. Crnković, "Component-based approach for embedded systems," in *9th International Workshop on Component-Oriented Programming (WCOP)*, June 2004.

[2] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković, "A Component Model for Control-Intensive Distributed Embedded Systems," in *11th International Symposium on Component Based Software Engineering*. Springer, 2008.

[3] E. Borde and J. Carlson, "Towards verified synthesis of ProCom, a component model for real-time embedded systems," in *14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE)*, June 2011.

[4] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A component model for secure microkernel-based embedded systems," *Journal of Systems and Software*, vol. 80, no. 5, 2007.

[5] K. Balasubramanian and D. C. Schmidt, "Physical assembly mapper: A model-driven optimization tool for QoS-enabled component middleware," in *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008, pp. 123–134.

[6] E. Borde, L. Pautet, and G. Haïk, "A new design approach for adaptative embedded systems," in *2nd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2009.

[7] C. Canal, P. Poizat, and G. Salaün, "Model-based adaptation of behavioral mismatching components," *IEEE Transactions on Software Engineering*, vol. 34, pp. 546–563, 2008.

[8] O. Gilles and J. Hugues, "Towards model-based optimisations of real-time systems, an application with the AADL," in *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE Computer Society, 2009, pp. 129–134.

[9] A. Aleti, S. Björnander, L. Grunske, and I. Meedeniya, "ArcheOpterix: An extendable tool for architecture optimization of AADL models," in *ICSE Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOPES)*. IEEE Computer Society, 2009, pp. 61–71.