Automatic Synthesis and Integration of Gray-box Components for Critical Embedded Systems — Reuse vs. Optimizations

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, the integration of software components requires development of code controlling their interactions. When this code is automatically produced from an architectural specification, a systematic approach where fully reusable code is generated for all entities in the system, leads to an overhead that is unaffordable in most embedded systems, as a consequence of temporal constraints and resource limitations. 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 facilitates 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. As a consequence, the developer can implement and evaluate more easily different options of reuse/optimization.

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

I. INTRODUCTION

More and more products in our everyday life take advantage of the miniaturization of electronics to provide functionalities that are controlled by a software embedded system. 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 even shorter time-to-market. Besides, developing computer systems that are embedded in cars, air planes, military systems etc., called *critical* embedded systems because a failure of such a system may have catastrophic consequences, brings another dimension of complexity by requiring dependability concerns to be considered as well.

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.

When integrating software components, wrapper code must be added to cope with this lack of knowledge and enable the components interactions, either with other components or with the underlying execution platform.

In a model driven development approach, integration code can be automatically produced by a synthesis process consisting of a set of model transformations and code generation steps that aim at producing binary images from a model. Considering this definition, the synthesis process answers to needs that emerge lately in the development process. Indeed, synthesizing the code aims at

1) testing the basic functionalities of an application (unit test and test of functional chains);
2) evaluating the non-functional properties of parts of the application (or the application itself); and
3) optimising the produced code in order to ensure the respect of non-functional properties.

Besides, synthesizing those binary codes requires to possess the implementation of the corresponding functional blocks, or a precise enough description of those functionalities from which the corresponding code can also be generated.

Although gray-box component models have been successfully used in general purpose software engineering, their adoption in Distributed Real Time and Embedded Systems (DRTES) still raises important challenges [1].

In particular, DRTES often have heterogeneous non-functional constraints, which makes the reuse of existing code in different usage contexts difficult. Indeed, generating code to produce a fully reusable entity requires considering that this entity might be reused in any usage context, thus leading to a systematic code generation process producing an unaffordable overhead. 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: the components reusability.

Answering the trades-off between those two opposite objectives is a difficult task since it requires the domain expertise to decide of the level of reusability or optimisation suitable for a given component.

In this paper, we present a component-based framework that automates the integration of reusable gray-box components. This frameworks helps in realizing the trade-off between reusability and optimization by proposing *i)* an architecture of generated code that separates clearly three different levels of reusability/optimisations, *ii)* a general purpose implementation of the generated code that aims reusability of the produced entity, and *iii)* optimization algorithms that proceed to the fusion of hierarchically nested components.

The remainder of this paper is organized as follows: Section II gives an overview of the contribution of this paper. In Section III we present ProCom, the component model our approach is based on. Sections IV, V, and VI present respectively the architecture of the generated code, the principles of its implementation in a reusability objective, and the optimization steps that lead to adapt this implementation to a particular usage context. Section VII describes how the proposed approach has been implemented in the ProCom development environment. Finally, we compare our contribution to related works in Section VIII, before Section IX concludes the paper and presents the perspectives of the contribution.

II. Overview of the Contribution

An important aspect of synthesis in a CBSE context is that it must facilitate reuse of existing functionalities while at the same time offering optimisation capabilities that enable to ensure that the final implementation respects the system's non-functional requirements. In the domain of real-time and embedded systems, optimization deals with reducing the memory footprint, the energy consumption of the whole system, improving the end-to-end performances and so on. On the other hand, reuse focuses on the integration, without any modification, of already designed and/or implemented entities.

To some extent, reuse and optimization are conflicting objectives, since the production of reusable units implies considering them independently of a particular usage context, while optimisation needs to take advantage of specific usage context information in order to improve the characteristics of the produced code. The approach presented in this paper provides the developer with an increased flexibility when addressing the trade-off between the two objectives.

We first address the synthesis of units that on one hand support reuse of unchanged implementation, and on the other hand allow for some degree of adaptation when reused. We propose an architecture where each synthesised unit can be viewed as a *gray-box* component consisting of three parts; an *interface definition* part that defines the data structure of a given component, an *interface implementation* part controlling how data and control are transferred in and out of the component, and an *internal implementation* part that provides the actual functionality. The interface definition can be reused unmodified in any context. When only the internal implementation needs to be reuse, which also allows for reuse of for example analysis and unit test results associated with this implementation, the interfaces implementation can be tuned for each particular usage context in order to provide an optimised access to the functionality of the inner part.

We also consider the synthesis of complex components, or entire systems, composed from such reusable gray-box components (possibly in several levels of hierarchical nesting). In the general case, the developed system consists of a mix of newly developed components, and components that are reused in-house or provided by third-party developers. Thus, some of the constituent subcomponents exist only in the form of already synthesised code, some exist only as an architectural specification with code for the individual elements, and others exist in both of these forms. The proposed approach allows the developer to specify for each entity in the design if it should be *i)* reused and adapted; or *ii)* completely (re-)synthesised for this particular context.

Components for which the developer selects the first alternative are said to be *marked as reusable*. When applied to a component for which code exists, this means that the *internal implementation* of the component is reused unmodified. When applied to a newly developed component for which code does not yet exist, that component is first synthesised separately, resulting in a gray-box component which will be included in the current system, but also possibly reused in other systems in the future. In both cases, the *interface implementation* part of the component is adjusted to increase the performance in this particular usage context.

The second alternative is only possible for components that include the architectural specification, e.g., newly developed components or components reused in-house. No part of that component is reused unmodified, which allows for optimisations spanning several component boundaries.

The search for an appropriate trade-off between reuse and optimisation for different parts of the system will be guided by the non-functional requirements of the system. However, the decision of synthesising a component for reuse or for optimal efficiency cannot be automated, since it requires the domain expertise of the software designer to identify its reuse potential. What can be automated is the detection of when additional optimisations are needed in order to meet the non-functional requirements of the system under construction.

Figure 1 illustrates these different options and their impact on the generated code. When a component is independently synthesized, no information about its usage context is available. It is thus synthesized according to a general purpose synthesis that aims at enforcing the respect of the semantic in any usage context (see (1) in the figure). When this component is used in a specific context, the synthesis process can take advantage of available information to optimize the produced code. In case the subcomponent internal implementation needs to be reused, – or in case of a primitive component – only the interfaces implementation is replaced (see (2.i) in the figure). Otherwise, the component is completely re-synthesized to maximize the benefits of the optimization (see (2.ii) in the figure).

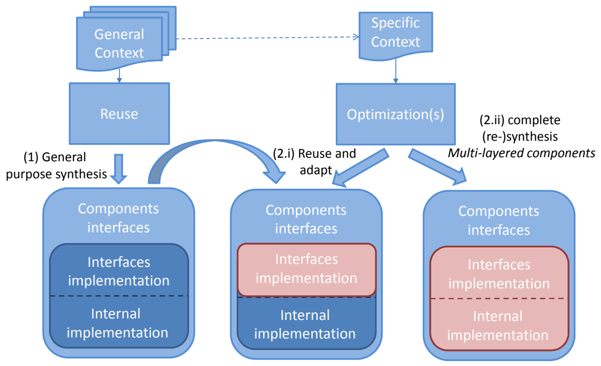

Figure 1. Synthesis process overview

We present in this paper a set of contributions that helps in automating this approach: we define the architecture for the generated code, we describe the principles of its general purpose implementation, and we propose a set of optimization steps. We also present in this paper a first prototype implementing these contribution. As a first evaluation of this prototype, we have used it to develop a simple application from the automotive domain. The first results we obtained are also given in this paper. Before to present those contributions in more details, we describe in next section the component model our framework relies on.

III. ProCom, the component model

The flexible 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.

A. General presentation

To address the different concerns that exist on different levels of the design of such systems, ProCom consists of two distinct, but related, layers. At the upper layer (namely ProSys), the system is modelled as a number of active and concurrent subsystems, communicating by message passing. Our synthesis approach, however, is mainly concerned with the lower level, called ProSave, which addresses the internal design of a subsystem down to primitive functional components implemented by code.

ProSave 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.

ProSave 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. At the highest level of nesting in ProSave, connectors representing periodic and aperiodic activation are used to turn a collection of passive ProSave components into an active entity that can be further modelled at the upper layer.

Figure 2 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.

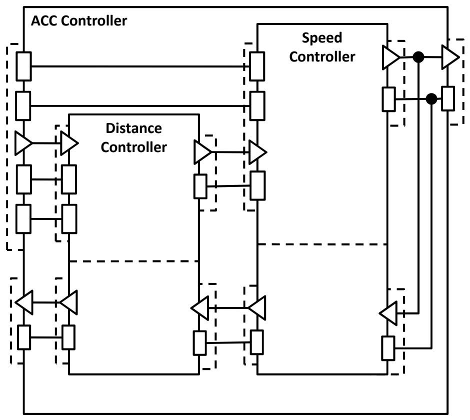

Figure 2. Composite ProSave component

B. Specificities of ProCom

A main characteristic of the component model 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.

In ProSave, ports are structured into groups consisting of one trigger port and a number of data ports, indicating that those data are always produced or consumed together, in an atomic and conceptually instantaneous action. The trigger port of a group is used to control when the action occurs.

Moreover, the functionality of a ProSave component is captured by a set of services, each consisting of one input port group and a number of output groups. The services of a component are triggered individually and can execute concurrently, while sharing only internal state data. The data at the input port group is accessed at the very start of each invocation, as a result of its trigger port (and thus the service) being activated. Any subsequent writing of data to the input ports will not made available to the component until the next invocation. Similarly, data written to the output ports during the internal computation performed by the service, will not become available to the rest of the system until the corresponding group is triggered. Before the service returns to idle, each of the associated output port groups must have been activated exactly once.

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.

As can be seen in Figure 2, port groups are denoted by dashed boxes framing data and trigger ports, and the services are separated by dashed lines.

C. Deployment of ProCom components

Modelling the deployment of ProCom components just consists of representing the allocation of ProSys components onto *virtual nodes* that are latter on mapped onto the concrete hardware platform. This information is then used to realize the deployment of components. Since ProCom is dedicated to the design of critical distributed real-time and embedded systems, components are deployed statically: the definition and initialization of data structures corresponding to components, tasks and interactions, is made at compile time (or during the very beginning of the system initialization if necessary). As a consequence of this choice, the definition of these data structures, as well as their initialization, is synthesised into the code of the system implementation.

This synthesis process mainly consists of the components-to-resource allocation. In the scope of distributed and real-time embedded systems, this allocation consists of mapping:

- interactions of ProSave components to shared variables and operation call sequences;
- ProSave components activation (clocks and communication channels) to real-time tasks;
- interactions of ProSys components to the physical communication media.

In this paper, we focus on issues related to the implementation of interactions between ProSave components. In order to implement the trade-off between reusability and optimizations, we propose a dedicated architecture for the generated code. This architecture is described in next section.

IV. Architecture of the Generated Code

The architecture of the generated code aims at maximizing the reuse of already generated code. To achieve this objective, we have identified the possible variations in the implementation of the ProCom semantic. These possible variations, as well as the corresponding impacts on the components wrappers implementation will be presented in Sections V and VI.

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 generated, and possibly validated. The basic principle of this architecture is to externalize every variable part in a *service handler* implementation, while the interfaces definition and the internal implementation represent the stable view of the service implementation.

Figure 3 represents the data structure generated for a 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 (enclosing part of the component representation in Figure 1). The second part represents the implementation of those interfaces, that can be adapted to a specific context in case only the interfaces and internal implementation of the component have to be reused (step 2.i in Figure 1). Finally, the third layer represents the internal implementation of a component. This last layer can be adapted when the usage context of this component is refined and only if the corresponding component can be completely re-synthesized (step 2.ii in Figure 1).

The interface definition is actually the data structure interfaced with the user code (code implementing the basic functionalities of the system). This architectural layer consists of two data structures, that reference each other: the *Component* data structure represents a component instance, as defined in the ProCom model provided by the software designer; the *Service* data structure corresponds to a service of this component. It is thus natural that a component reference several services while a service references only one component. Besides this mutual references, those data structures contain additional information:

- the *Component* data structure defines a reference to the internal state of the component, a data structure that can be used by the functional code to define, initialize, and use some internal state;

- the *Service* data structure also defines some reference to its input and output ports.

In Figure 3, the interface implementation corresponds to the variable part of the generated code. The *Service_handler* data structure is the main constituent of this layer. This entity implements the context independent aspects of the service interfaces, and will be replaced when the usage context is known. It references the handled service and the executed sub-services handlers in order to implement the interaction between components.

Finally, the *Subservice_handler* data structure corresponds to the internal implementation of a composite component. For a primitive component, the internal implementation is directly given by the user code. For a composite component, this entity gathers the definition of subservices, either in their adapted or reusable definition or in their re-sunthesized definition (depending on the possibility to adapt the corresponding subcomponents). In case only the interface definition of a component has to be reused, this internal implementation will be re-synthesized when refining the usage context of this component.

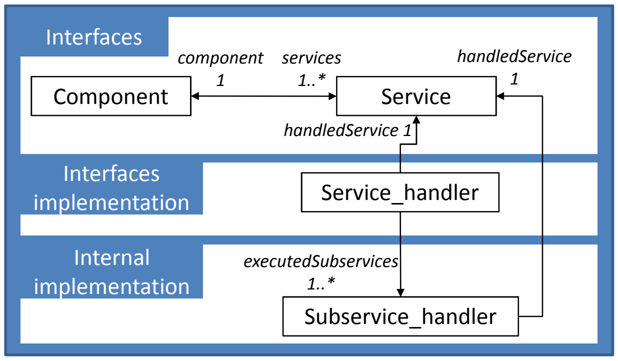

Figure 3. Generated Code Architecture

This architecture enables to identify the variable and stable parts of the generated code. We present in next section the principle that led the implementation of this architecture in a general purpose synthesis.

V. General Purpose Synthesis

The first objective of a synthesis process (see Section I) is to test the basic functionalities of an application. Those basic functionalities are identified in ProCom as software components. When independently synthesized, the synthesis tool does not have any information about the context in which these components 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.

Some aspects of this semantics were summarized in Section III, and we extract from this specification the following properties that have to be ensured in the generated code:

1) Service locks: if an input trigger port is activated while the corresponding service is active, the new activation is ignored.
2) The data ports of an input port group are all read together in an atomic action, performed when the trigger port of the group is activated. This data must not be modified until the end of the service execution.
3) When an input trigger port is activated, the component execution *must* trigger each of the output port groups of the corresponding service once and only once. In case of multiple output port groups, this trigger out action does not have to be performed at once for all of them.
4) Data corresponding to output data ports in the same group are all transferred to the connected components in an atomic action, performed when the trigger port of the port group is activated.
5) Connections involving connectors represent multiple interactions, each of them enforcing the same semantics of a unique direct connection.

A. A Three Phases Behaviour

Figure 4 represents the data interaction implementation between two components in the general case. Both a general case specification, and a schematic realization of its three phases implementation, are illustrated on this figure.

General Case Specification: The specification of a general case is illustrated on the top of Figure 4. In this specification, two components (A and B) are triggered by two clocks with different frequencies (75 and 100 Hz). The control flows of those two components are independent, while their data flow share a simple connection.

Three Steps Implementation: Considering such a specification, the ProCom semantic requires implementing a three steps behaviour, illustrated at the bottom of Figure 4:

1) When the input trigger port of a service is triggered (*TriggerIn_A* in the figure), this service switches from an idle state to an active state in which it ignores any input triggers, thus enforcing *property 1*. Then for each input data port (*d_in_A*), the service implementation transfers the data received on this data port connection (*cnx_d_in_A*) into an internal instance of this data (*d_in_A* at the bottom of the figure), stable until the end of the service execution (partially enforces *property 2*, atomicity issues will be treated independently in the remainder of this section);
2) In the second phase of the service behaviour, the reusable functionality (for which we do not know the exact behaviour) computes the output (*d_out_A* at the bottom of the figure) of the service thanks to the received inputs (*d_in_A* at the bottom of the figure). The service wrapper checks if all the output port groups have been triggered, and resume the component in case of pending computations (see *property 3*); the

wrapper also checks if at least one of the output has been triggered (see *property 3*). Note that writing the values of the data during this phase does not mean that the corresponding data is actually transferred.

3) Finally, when the output trigger port (*TriggerOut_A*) of the service is triggered, the service implementation produces the output data to be read by the connected input data ports (*property 4*); and transfers the control to the connected trigger ports.

The way interactions are implemented in case of multiple connections is treated latter in this section.

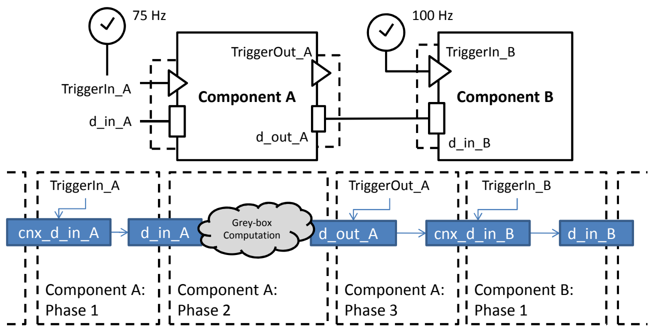

Figure 4. A Three Step Interaction Mechanism

As can be seen in Figure 4, the general case requires dealing with shared data accesses. Indeed, in the figure, one can easily see that the data representing the interaction between A and B (*cnx_d_in_B*) might be accessed concurrently by the corresponding services. In next section, we explain how we protect this shared data access to preserve the ProCom semantics and more precisely the atomicity of component interactions.

B. The Double Buffer Interaction

In this section, we tackle the issue of atomicity in data flow interactions (*property 2* and *4*).

The main hypothesis of data transfer between ProSave components is atomicity. Of course, this hypothesis cannot be ensured in a multi-threaded implementation since it requires data copying and locking. However, the implementation must ensure that the data transfer pattern (emission and reception) is the same as the one that has been considered for analysis [3]. To ensure this, we propose to rely on a double buffer implementation for data interactions: the output port of a component is deployed as a set of two buffers, one that can be updated during the execution of the component, and one that contains the last up-to-date value. The overhead of this solution is a pointer that references the last up-to-date value (thus switching from one buffer to the other one when the data is transferred out of the writer component).

In order to ensure data consistency, the first and third steps of an interaction (respectively transferring the data inside the reader and outside the writer) must never be simultaneously accessed. To ensure this, the double buffer solution has been preferred to a single buffer implementation since it reduces time spent in the critical section (only a pointer switch on the writer side), thus getting closer to the atomic hypothesis. Besides, it is similar to the solution used in the scope of synchronous programming, for which a semantic conformance proof has been provided [4]. However, we did not yet prove that this implementation conforms to the hypothesis of the ProCom model analysis [3]. As one can easily understand, the conformance between this implementation and the analysis semantic is very difficult to ensure.

C. Multiple Interactions

One of the main advantage of the ProCom component model is to model explicitly the control and data flow of components. However, the semantic of ProCom components only enables to define a partial control and data flow. Indeed, if a ProCom component has several output port groups, the semantic states that for each activation of a service, each of the output port group:

- will be triggered once and only once;
- can be triggered at different points in time.

As a consequence, the order in which ports are triggered is not known at design time (reason why ProCom components are qualified of "gray-box" components in this paper). Thus, the generated code must enable the orchestration of components in functions of the trigger information received at runtime. To deal with this property, the upper level component stores the activated input ports in a list and schedules the corresponding subcomponents according to a predefined orchestration policy. In case of adaptation of this orchestration to a particular context, only this part of the component implementation (belonging to the *interfaces implementation* layer in Figure 3) would be impacted.

Another important aspect of the ProCom component model is the existence of connectors. Such entities lead to consider multiple interactions in which an output trigger port activates several input ports. In such a situation, *property 5* states that the atomicity of interactions must stand for each interaction. As a consequence, the generated code first transfers the trigger and data to all the connected ports (in one atomic step), and then executes the inner implementation of the activated sub-services. Those sub-services are ordered according to a predefined policy.

This orchestration specification might be refined functions of non-functional requirements such as timing properties and memory constraints. Without such specification, the context independent synthesis chooses an execution arbitrarily.

Besides the ordering of the components execution, different parts of the implementation presented in this section might take advantage of a better knowledge of the component's usage context. Following this idea, we present in next section the optimizations steps we propose to reuse and adapt synthesized units.

VI. Reuse and Adaptation

The general purpose synthesis produces an implementation which is guaranteed to satisfy the semantics of the component model independently of the enclosing system. This general implementation provides a means to work with the component in isolation, for example to perform unit testing or measure properties of interest. However, whenever the component is used (or reused) within a larger system, the external part of the component implementation can be adjusted based on information about that particular context, to reduce the overhead.

The components usage context is refined all along the design process. Generally speaking, it consists of all entities in the 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.

The main 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 component to synthesise – and it should be noted that this could be an entity at any level of hierarchical nesting, for example a component that was marked *reusable* in the synthesis of some component even further up in the hierarchy – 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 IV. In summary, the different steps of the synthesis algorithm are the following:

1) Flatten (down to primitive components, or components marked as *reusable*) to a single non-hierarchical model.
2) Extract the control and data-flow of the different services.
3) Refine, based on the existing non-functional requirements, the order of service execution.
4) Determine where service locking must be enforced to avoid re-entrance.
5) Remove port groups that are not needed for correct synchronization.
6) Decide for each data producing port whether full locking and buffering is needed.

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

A. Model Flattening

The model flattening is completely safe with regards to the preservation of the components semantic. This step consists of representing the whole tree of nested subcomponents as a single collection of connected primitive and *reusable* components. The composite components that are not marked *reusable* are removed, and their internal subcomponent structure is added to the enclosing level, together with the input- and output port groups of the removed composites. These *orphan groups* are needed, in the general case, to ensure correct synchronization in the flattened model, but in many cases they also will be removed (in step 5).

Figure 5 illustrates this model transformation on a simple example. The composite component B is removed, resulting in two orphan port groups, but component C remains since it is marked as *reusable*.

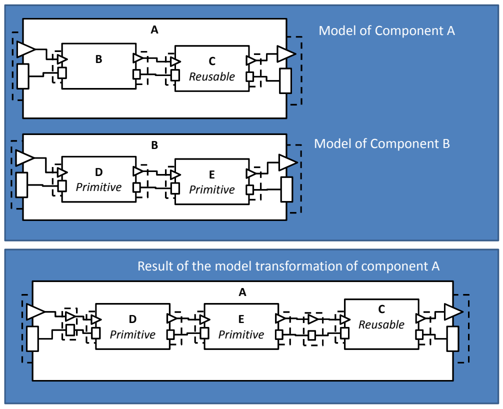

Figure 5. Illustration of the first model transformation step.

B. 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. The target meta-model of this transformation is given in Figure 6. The root entity of this meta-model is a *Composite* component consisting of a number of composite services (*CompositeService*). Each composite service defines a set of sub-services that will be executed in the scope of the synthesized entity. In addition, a composite service defines an entry point (*EntryPoint*), representing the main function of the synthesized entity. This entry function references a set of progress steps (*ProgressStep* with its self reference), that represent the different advancement points in terms of control flow. Depending on the type of progress step (*ProgressStep* is an abstract class), the associated action in the generated code will be different. For instance, an *EndPoint* entity represents one of the trigger output port of the synthesized entity.

C. Control Flow Refinement

The control flow defined in the component model can contain fork- and join connectors, which means that the branches after the fork can be executed in any order, possibly interleaved, but that the path after the join must wait until all fork paths have been fully executed. In this step, such non-determinism is transformed into a fixed orchestration by selecting one of the permissible execution orderings. The component model also allows dynamic selection of execution paths, but since this cannot be resolved during synthesis, the possible outcomes are all represented in the graph.

Another source of non-determinism in the model is subcomponent services with multiple output ports, meaning that they can produce output data at different points in time. Since the internals of the subcomponent is off-limit for the synthesis, all possible orders in which the groups can be activated have to be represented as a dynamic selection. Moreover, a decision must be made regarding at what point in the orchestration the subcomponent should be resumed, allowing it to produce the data of the remaining output groups.

Without any requirements on the synthesised component, it is not possible to make wise choices about the optimal ordering. 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.

D. Removal of Service Locking

The removal of a service lock is possible if it can be determined that the corresponding port can never be triggered when the service is already active. In fact, the only situations in which a service lock must be kept are when *i)* the service can be reached from more than one service in the synthesised component; or *ii)* the component is found at the top level of ProSave and can be activated in several tasks that cannot be merged using the techniques presented in [5] (which also assumes that the task model forbids tasks re-entrance).

E. Orphan Port Group Removal

An orphan port group G can be removed if the following criteria hold in the refined control- and data flow model:

i) all incoming data connections to G come from ports in groups that are never activated after G; and

ii) all outgoing data connections from G lead to ports in groups that are never activated before G.

It should be noted that "never activated before/after" requires that the two groups belong to the control flow of the same composite service. If this is not the case, the relative ordering of them is unknown.

When an orphan port group is removed, each incoming connection to a port in the group and the corresponding connection leading out of the port are replaced by a single connection.

F. Removal of Data Locks and Double Buffers

The removal of data locks is a bit more complex. The purpose of data locks is to avoid concurrent reading and writing of data, since this could lead to partially written data being read, which would be a violation of *property 2* and *property 4* stating that data transfer and writing are atomic actions. However, if all writing and reading of a particular data are performed in the execution of the same parent service, or in a unique task (assuming that the task model forbids tasks re-entrance), then there can be no concurrent accesses, and the lock is thus not needed. Once more, merging tasks thanks to techniques such as [5] are of great interest in this case .

The role of the double buffers is to ensure another aspect of *property 2* and *property 4*, namely that data is forwarded only when the trigger port of the group is activated. The condition under which this can be ensured without an additional buffer for the transfer of data from a port in group G_p to a port in G_c can be defined as follows:

i) G_p and G_c belong to subcomponents, not to the synthesised entity.

ii) G_p is either always triggered before G_c, or always triggered after the full execution of the sub-service that G_c 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 components are triggered and when output ports groups are triggered.

VII. From Specification to Binary Code Production

In order to evaluate the approach presented in this paper, we have implemented a synthesis framework that automates the static deployment of ProCom components. This framework implements most of the optimization steps presented in Section VI, and produces C code according to the architecture presented in Section IV. We present in this section the architecture of this framework, as well as the code generation work-flow. This work-flow is summarized

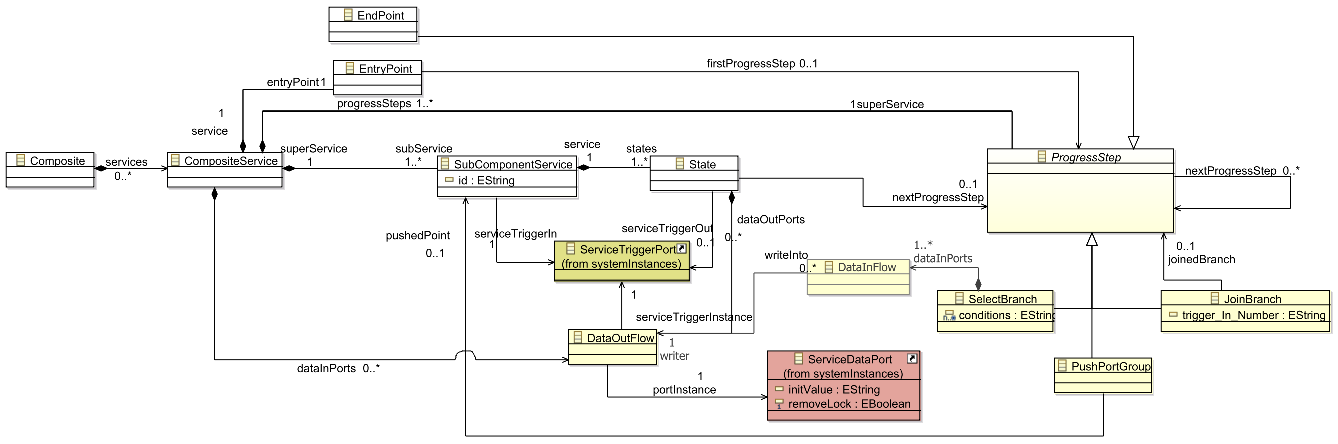

Figure 6. Control and Data Flow Meta-Model

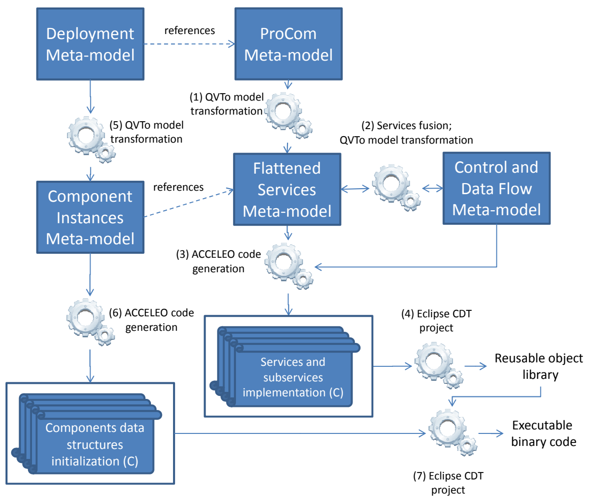
Deployment Meta-model
references
ProCom Meta-model
(1) QVTo model transformation
(5) QVTo model transformation
(2) Services fusion; QVTo model transformation
Component Instances Meta-model
references
Flattened Services Meta-model
Control and Data Flow Meta-model
(3) ACCELEO code generation
(6) ACCELEO code generation
(4) Eclipse CDT project
Services and subservices implementation (C)
Reusable object library
Components data structures initialization (C)
Executable binary code
(7) Eclipse CDT project

Figure 7. Components Generation Work-flow

in Figure 7, which illustrates both the components synthesis and integration (through the deployment part) work-flow.

The architecture of the synthesis tool is made of a set of Eclipse plug-ins that make an intensive use of the facilities of this platform in terms of model-to-model transformation (QVTo[1]), code generation (ACCELEO[2]), and C code edition and makefile generation (Eclipse CDT[3]).

[1]http://www.eclipse.org/m2m/
[2]http://www.eclipse.org/acceleo/
[3]http://www.eclipse.org/cdt/

A. From Components Implementation to Reusable Objects Libraries

The components implementation and build work-flow consists of 4 steps (1 to 4 in Figure 7):

Step 1 consists of flattening the hierarchical component model down to primitive components or components marked as reusable. This is implemented as a QVT transformation from the ProCom meta-model to an intermediate representation defined as an EMF meta-model as well. During this step, the elements of the source meta-model are simply copied in the target meta-model. Thus, the difference between the source and target models is that the source meta-model only contains one layer of composition at a time by referencing the definition of the subcomponents, while the target meta-model contains the final representation of components instances. This transformation step implements the flattening of ProCom model presented earlier (see Section VI-A and Figure 5).

Step 2 also consists of a QVT model-to-model transformation. The source model is the result of the first step while the target model is a representation of the (possibly incomplete) control and data flow specification. From this result, the source model is also modified to represent the service fusion presented in Section VI (removing service and data locks where possible).

During step 3, the control and data flow meta-model is used as an entry point of a code generator (using ACCELEO). From this meta-model, the complete control and data flow of the synthesized service are generated: this includes the realization of connectors, and the orchestration of components, as well as the implementation of the data transfers.

Finally, step 4 is made of a customized CDT plug-in that generates the makefile and builds a static library of the synthesized entity.

B. From Virtual Nodes to Components initialization

The virtual node implementation and build work-flow is made of 3 steps (5 to 7 in Figure 7).

The entry point of the first step is the deployment model that associates a set of ProSys subsystems to an execution platform. From this deployment model, the tree of component instances deployed on this execution resource are gathered in an intermediate model (instances model on the figure) that references the flattened services models of the entities to be deployed. This reference enables to retrieve information regarding the optimizations that had been performed in earlier phases of the generation work flow.

VIII. Related Work

The contribution presented in this paper is a component framework that automates the synthesis and integration of reusable components designed as sub-functionalities of embedded systems. In order to respect the system's non-functional requirements, the implementation of those components have to be optimized according to their usage context. The framework we presented in this paper helps in answering the trade-off between reusability and optimization by proposing a generated code architecture that identifies and isolate the reusable code from the adaptable code.

Considering the scope of this contribution, the related works are mainly twofold:

1) frameworks for reusable components in embedded systems;
2) model-driven or architecture-based optimizations for embedded systems.

A. Frameworks for Reusable Components in Embedded Systems

Many component frameworks already exist both in industry and in academic research. Among those, we have selected those that we thought were more closely related to the contribution presented in this paper. This comparison is not exhaustive, but gives an overview of the originality of the contribution presented in this paper.

To begin with, we have selected only component frameworks for which components are grey-boxes: their internal behaviour is not exhaustively known. This selection excluded the BIP [6] component framework and frameworks based on synchronous languages. Besides, we have focused on component models and frameworks that aim the synthesis and integration of reusable components in embedded systems.

CAmkES [7] 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. The two firsts are commonly used in most component models, while the last is more specific to this particular component model. 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.

PECT [8] is a component framework that focuses on the usage of analytical theory to enforce predictability of non-functional properties by construction. As a consequence, this component framework does not focuses on the integration and synthesis of component. In this framework, the interaction semantics are *i)* event-based with message queues or *ii)* synchronous with the possibility to define the necessity for a protected critical section. Thus, the usage of locks is decided at design time.

THINK [9] is a component framework that specifically targets the development embedded telecommunication systems. The components interaction paradigm is mainly interface based.

CIAO [10] and MyCCM-HI [11] is a component framework dedicated to the adaptation of the Lw-CCM[4] standard to the domain of real-time and embedded systems. The scope of the work presented in [10] 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 [11], the authors use a static deployment and configuration approach on the 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 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*.

B. Model Driven Optimizations for Embedded Systems

In the scope of model driven optimizations for embedded systems, two works have particularly caught our attention.

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

Both use AADL[5], a component-based language in which the modelling artefacts represent concrete entities of the software and hardware architecture (processes, threads, data, subprograms, processors, physical memory, etc.). Besides, AADL defines a precise execution semantic for each of the software components that enables the formal analysis of the whole system.

In the scope of the development of the Ocarina tool suite, that aims at synthesizing AADL model into different programming languages [5], proposes an approach to fusion the different activities (or thread) of a real-time embedded system. As we already stated earlier (see Section VI), this work is a complement to the contribution presented here. Indeed, this fusion of activities can be an interesting result to be used as an input of our optimizations.

In ArcheOpterix [12], 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 functional blocks. In [5] and [12], the focus is on optimizations at a different level of abstraction, closer to the final realization of the overall system.

IX. 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 component framework presented in this paper automates the production of the code implementing the components integration. This implementation relies on an architecture specification that enables to identify clearly its reusable and adaptable parts. Besides this framework and the architecture of the generated code, we also provide a description of the components integration code that targets the respect, in the general case, of a formally defined semantic. Last but not least, 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 a first experimental result, the usage of these optimizations led, on a simple example, to a reduction of 30% of the size of the produced binary code.

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

These promising results led us to consider different perspectives to this work. Firstly, a more extensive experiment would enable to evaluate deeper our framework and particularly our optimization algorithms. Secondly, an interesting question is to evaluate if optimizing the generated code from the component-based specification could lead to optimizations that the compiler could not do later on because of complex dependencies between the control and the data flow. Finally, 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.

Acknowledgment

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre Progress.

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 Berlin, October 2008, pp. 310–317.

[3] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu, and P. Pettersson, "Formal semantics of the ProCom real-time component model," in *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, August 2009.

[4] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis, "Semantics-preserving multitask implementation of synchronous programs," *ACM Transactions on Embedded Computing Systems*, vol. 7, pp. 15:1–15:40, January 2008.

[5] 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.

[6] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-time Components in BIP," in *Proc. of the 4th IEEE International Conference on Software Engineering and Formal Methods*. IEEE, 2006, pp. 3–12.

[7] 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, pp. 687–699, 2007, special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems.

[8] K. C. Wallnau, “Volume III: A Technology for Predictable Assembly from Certifiable Components (PACC),” Carnegie Mellon, Tech. Rep. CMU/SEI-2003-TR-009, 2003.

[9] O. Lobry, J. Navas, and J.-P. Babau, “Optimizing component-based embedded software,” in *2nd IEEE International Workshop on Component-Based Design of Resource-Constrained Systems (CORCS)*, Jul. 2009.

[10] 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 (RTAS)*. IEEE Computer Society, 2008, pp. 123–134.

[11] 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.

[12] A. Aleti, S. Bjornander, 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.