37th EUROMICRO Conference on Software Engineering and Advanced Applications.
August 30 to September 2, 2011. Oulu, Finland.

Flexible Semantic-Preserving Flattening of Hierarchical Component Models

Thomas Lévêque, Jan Carlson, Séverine Sentilles
Mälardalen Real-Time Research Centre,
Mälardalen University, P. O. Box 883,
SE-72133 Västerås, Sweden
Email: firstname.lastname@*mdh.se*

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

***Abstract*—Hierarchical component models allow to better manage system design complexity compared to flat component models. However, many analysis techniques lack support for dealing with hierarchical models. This paper presents a general approach to use existing analysis on hierarchical component systems by means of a flattening transformation. The transformation can be partially applied, which provides a possibility for tradeoffs between analysis scalability, result precision and reusability concerns.**

The general approach has been implemented and evaluated in the context of ProCom, a hierarchical component model for real-time embedded systems. As a result, the paper describes a flattening transformation which preserves the ProCom operational semantics and presents the related optimizations.

***Keywords*-component model; transformation; analysis;**

I. Introduction

Component-based software engineering (CBSE), where systems are built from pre-existing components interacting though well-defined interfaces, has been recognized as a promising approach to deal with complexity and facilitate reuse. To address the complexity of modern software, *hierarchical component models* support components that are in turn built from a collection of interconnected smaller subcomponents. Contrasting these *composite* components, the *primitive* components at the bottom of the hierarchy are implemented by code.

The use of CBSE also brings other benefits, including the possibility to model the functionality and extra-functional properties (EFPs) of individual components, in order to establish system level properties at an early stage of the development. This is particularly important in domains where aspects such as dependability and timing are crucial to the overall correctness of the system, for example in critical real-time embedded systems.

In the context of hierarchical models, we distinguish between two classes of analysis methods that operate on a single hierarchical level. *Compositional analysis* can be applied recursively to each composite entity in the hierarchy, since the result from one serve as input to the analysis of the enclosing entity. As an example from this category, we can consider analysis deriving the worst-case execution time (WCET) of a composite component, based on the WCET of subcomponents and the way they are interconnected.

Contrasting this, *non-compositional analysis* cannot be applied recursively, e.g., because the analysis does not produce the same information about the composite entity as it requires as input for the subcomponents. To exemplify this category, we can consider an analysis that checks if the composite component is deadlock-free, based on input-to-output dependencies of the subcomponents.

Applying non-compositional analysis techniques to hierarchical component models is difficult and requires a lot of additional manual effort. For example, transforming manually the hierarchical model into a flat variant accepted by the analysis, is error-prone and can make it hard to locate problems identified by the analysis back to the right component in the original model. At the same time, restricting developers to flat component models limits their ability to handle complexity during the development process.

Compositional analysis, on the other hand, can be directly applied to hierarchical models. However, this typically comes at the cost of reduced analysis precision, since the information propagated between hierarchical levels is often a subset of the information available during the analysis of a single level. For example, the compositional WCET analysis outlined above is based on the WCET of subcomponents, but has no information about the conditions under which these worst cases occur, and must thus consider as a safe over-approximation the case when all subcomponents exhibit their worst possible behaviour at the same time.

This paper proposes a generic approach to allow both compositional and non-compositional analysis to be used on hierarchical component models, using a flattening transformation which preserves the original operational semantic. In the case of compositional analysis, the approach provides increased flexibility by facilitating tradeoffs between scalability, precision and reuse concerns. The approach has been experimented in the context of ProCom, a component model specifically targeting real-time embedded systems [1].

Section II gives a high-level description of the proposed solution, followed by a detailed concretization of the approach for the ProCom component model in Section III. Section IV presents results of our experiments on ProCom, and related works are described in Section V. Finally, Section VI concludes and presents ongoing and future work.

II. The Approach

As stated in the introduction, many analysis techniques cannot be performed directly on hierarchical component models. In order to benefit from the EFP computation capability of these existing analyses as well as benefiting from the abstraction and reuse advantages of hierarchical component models, our main objectives are to:

i. make the integration of existing analysis on hierarchical component models as easy as possible; and
ii. deal with analysis scalability and result precision.

The main idea of our approach is not to change analysis but to work on the analysis inputs to reduce their complexity and their size. Our generic approach addresses the considered objectives with following solutions:

- Instead of modifying analysis, we choose to transform a system described by a hierarchical component model to a model which can be used as input to existing analysis. This is achieved by providing a *model transformation* which flattens all composition levels (objective *i*).
- One recurrent problem with analysis is their scalability and their execution time. To speed up the analysis and improve their scalability, we use *optimization* to reduce model size by removing details that are redundant for the current analysis purposes (objective *ii*).
- When dealing with large systems, scalability remains the major problem for analysis. For compositional analysis, we choose to perform analysis *incrementally* on different composition levels. To allow a tradeoff between scalability concerns, analysis precision and reuse of analysis results, the flattening transformation is *parameterized* to define which components to flatten (objective *ii*).

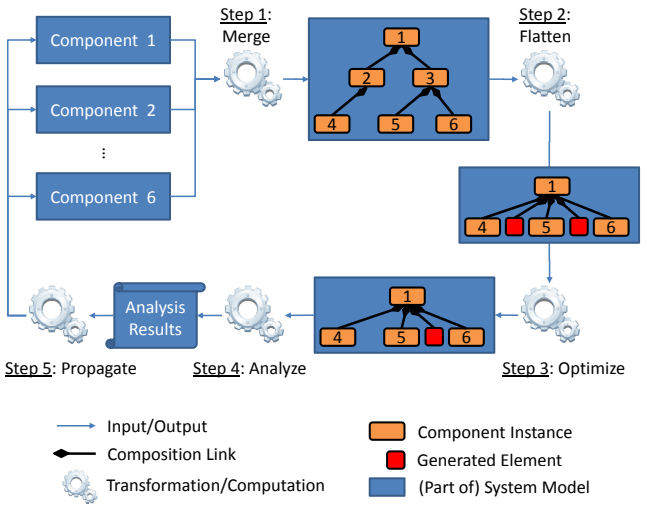

Figure 1. Basic Flattening Process.

Algorithm 1: Basic Flattening

Input: Component specifications for all primitive and composite components in the system.
Output: Analysis results and/or new EFP values associated with the component specifications.

Step 1: Merge
Merge all component specifications into one model with explicit links. If the individual component specifications contain implicit references to other entities, e.g., relying on naming conventions, these are replaced by explicit references.

Step 2: Flatten
Perform the flattening transformation. Informally, this is done by replacing references to a composite component by direct references between entities inside and outside, before removing the composite. In addition, some new elements might have to be introduced in order not to change the operational semantics of the system. Associate EFPs with the generated elements, if required by the analysis in question.

Step 3: Optimize
Optimize the flattened model. For example, some of the elements introduced during flattening might not be needed in this particular system or for this particular analysis, and can thus be removed.

Step 4: Analyze
Perform the needed analysis on the optimized flattened model.

Step 5: Propagate
If the analysis results in new EFPs of the analysed entities, propagate these to the original model elements.

A. Basic Flattening

The basic flattening approach is defined in Algorithm 1, and the process is illustrated by an example in Figure 1. In the example, starting from a system model consisting of six components (three composite and three primitive), the first objective is to merge them into a single model with explicit links between elements (Step 1). Then, the composite components 2 and 3 are flattened, resulting in a model consisting of a single composition level (Step 2). After that, the flattened model is optimized in order to reduce analysis time, and we assume in the example that one of the generated elements is identified as semantically redundant (Step 3). Then, analysis can be performed on the flattened model (Step 4), and finally, the computed EFP values are assigned back to the original model elements (Step 5).

The basic flattening process relies on the following assumptions:

1) A flattening transformation exists that preserves relevant properties.
2) The size of the resulting model does not exceed what analysis can handle.
3) Support exists for annotating the elements of the component model with extra-functional properties.

To preserve relevant properties of the original component model, the flattening transformation might need to add new model elements, such as components and connectors. Note that it is the analysis purpose, and especially the information upon which it relies, that should define what properties must

be preserved by the flattening transformation. Otherwise, the correctness of analysis result is not ensured.

In cases where the analysis technique is based on specific properties set on the model elements, these additional model elements have to be decorated as well with these properties to enable performing the analysis. Taking again the case of WCET analysis as an example, having the WCET values for each subcomponent is a prerequisite for the analysis, and thus this must be specified for any generated element.

Setting these values can be part of the flattening transformation when the considered properties are known. However, component models usually allow to define new EFP, e.g., as a pair of *key* and *value* only. This cannot be handled by a generic transformation, and we propose instead to associate to each new EFP type definition a specification of how EFP values of that type can be computed for the generated elements.

B. Flexible Flattening

The basic process presented above does not completely fulfil objective *ii* as analysis scalability problems are only addressed using some optimization techniques to reduce the size and complexity of the model. In case of compositional analysis, a complete flattening potentially increases analysis precision, but at the same time ruins the benefits of the compositional approach in terms of scalability.

In order to give developers the means to balance the benefits of flattening and compositional analysis, we propose to parameterize the flattening transformation. The new process is exemplified in Figure 2, following the steps defined in Algorithm 2. Unlike the original approach, flattening is performed on parts of the model instead of the whole model. To this end, Steps 3 to 6 are iteratively performed according to the components marked as partitioning points, i.e., components that we do not want to flatten.

Note that partitioning points do not have to be in separate parts of the hierarchy, as in the example. We might as well have selected 3 and 9 to be the partitioning points.

In Figure 2, components 3 and 4 are selected as partitioning points. The flattening process is driven by this choice, and the merged model is partitioned into three submodels corresponding to components 1, 3 and 4. Flattening, optimization and analysis are performed on these models separately, in a bottom-up order.

To ease the complex tasks of identifying good partitioning point to drive model splitting, we propose to use component properties to select candidates. For example, the following properties could be helpful to guide the selection:

- *Reusable:* Components intended to be reused are in general at a good granularity. They are not too fine grained neither too coarse grained. As reusable units, multiple instances of them can have been defined, so their analysis results can be reused instead of being recomputed multiple times.

Algorithm 2: Flexible Flattening

Input: Component specifications for all primitive and composite components in the system, and a set of partitioning points.
Output: Analysis results and/or new EFP values associated with the component specifications.

Step 1: Merge
Merge all component specifications into one model with explicit links.

Step 2: Partition
Partition the model such that the top node of a submodel is either a partitioning point or the top node in the merged model, and the leaves are primitive components or partitioning points.

For each submodel, in bottom-up-order:

Step 3: Flatten
Perform the flattening transformation.

Step 4: Optimize
Optimize the flattened model.

Step 5: Analyze
Perform the needed analysis on the optimized flattened model.

Step 6: Propagate
If the analysis results in new EFPs of the analysed entities, propagate these to the original model elements, and to the submodel where this top node occur as a leaf.

- *Substitutable:* A component can be intended to be substitutable. In this case, there should not be any impact on the system using this component if the internal realization is changed and the component external properties remain unchanged or are at least still valid. This implies that system analysis should rely only on the external properties of the substitutable components.

Other properties could also be considered, such as strong timing requirement which require strong optimization and accurate estimates. Metrics related to these properties could be automatically computed to help the developer find good partitioning point candidates. Further investigations must, however, be performed to evaluate the suitability of different metrics with respect to the partitioning points.

The potential benefit on analysis scalability can be observed in the refined assumptions:

1) Analysis is composable.
2) A flattening transformation exists that preserves relevant properties.
3) Analysis can handle the size of the largest submodel.
4) Support exists to annotate the elements of the component model with extra-functional properties.

On one hand, performing analysis on every composite component may be costly and may result in loss of result precision, such as overestimates. On the other hand, performing an analysis on the whole system can be infeasible for scalability reasons. That is why a trade-off between analysis execution time, scalability and result precision must be defined.

The rest of the paper describes an application of this approach to the ProCom component model. The main objec-

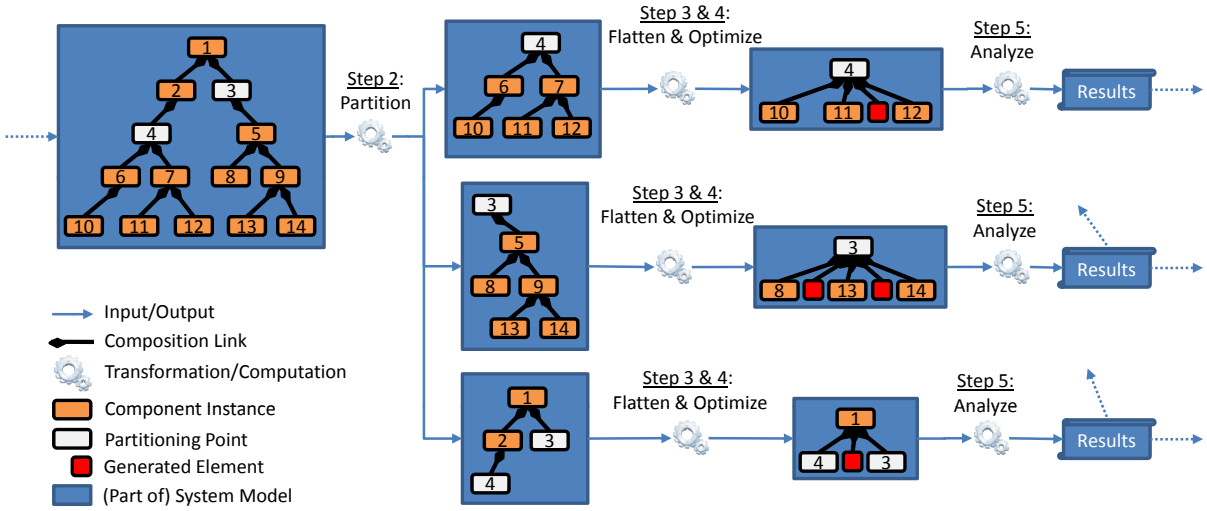

Figure 2. Flexible Flattening Process.

tive of the related experiments was to demonstrate that the approach can be applied. To this end, we define a flattening transformation which preserves the original operational semantic, present two optimizations which aim to reduce complexity and size of resulting flattened model, and show some performance results of the implementation.

III. Flattening in ProCom

In this section we concretize the general approach by applying it to a particular component model. Before defining the flattening transformation, we introduce the syntax and operational semantics of this component model.

A. ProCom Component Model

The ProCom component model [1] is specifically developed to address typical concerns in the domain of distributed embedded systems, such as resource limitations and requirements on safety and timeliness. ProCom is organized in two distinct layers that differ in terms of architectural style and communication paradigm. For this paper, however, we consider only the lower layer, where a system or subsystem is modelled as a collection of passive components based on 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. Data ports always appear in a group together with a single trigger port, and the ports in the same group are read or written together in a single atomic action. Together, an input group and its associated output groups are called a *service*.

Figure 3 (a) shows a simple ProCom component with one input port group and two output port groups (denoted by dashed lines). Triangles and boxes denote trigger- and data ports, respectively.

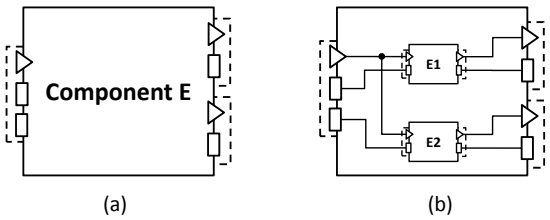

Figure 3. Example of a composite ProCom component, with (a) being its external view and (b) the internal view.

ProCom is hierarchical, allowing components to be internally constructed as a collection of interconnected subcomponents. *Primitive* components are implemented by C functions. The functionality of a *composite* component is determined by its subcomponents, their interconnections, and *connectors* that provide detailed control over the data- and control flow. The set of ProCom connectors include constructs for forking and joining data and trigger paths, and for dynamically selecting an execution path depending on current data port values. Figure 3 (b) exemplifies the definition of the internal contents of a composite component consisting of two subcomponents and a *fork* connector, graphically represented by a filled circle.

ProCom components do not communicate freely through their ports, but follow a strict read-execute-write cycle, where the activation is always initiated externally and the dependencies between input and output are explicitly modelled in the component interface. All services of a component are initially in an idle state, just receiving data on its input data ports. When a service is triggered, i.e., when the input trigger port is activated, it switches from idle to active state in which it ignores any further incoming triggers. The active phase consists of the following steps:

1) The data at the input data ports of the service are atomically copied to internal representations which remain unchanged until the end of the service execution.
2) The service functionality is executed. Updates to the output data ports of the service are not made visible externally until the trigger port of that port group is activated (from inside the service).
3) When all output port groups have been triggered once, the service immediately returns to the idle state.

B. The Flattening Transformation

Considering the overall approach presented in the previous section (see Figure 1), Step 1 is simplified in case of ProCom, since ProCom models contain only explicit links and do not rely on naming convention or other implicit link mechanisms.

In ProCom, composite and primitive components are subject to the same restrictions in terms of the read-execute-write cycle described above. Two aspects of the operational semantics are of particular interest in the context of flattening:

- *Service locking:* Triggering of an already active service is ignored.
- *Group synchronization:* Data ports of a group are always written or read together, in an atomic and conceptually instantaneous action.

Without these, flattening could be achieved by simply replacing a composite component by its internal structure, and merging incoming and outgoing connections of each port of the composite into a single connection.

In this section, we show how flattening can be performed in ProCom without violating the operational semantics, by introducing constructs that explicitly implement the service locking and group synchronization behaviours of the removed composite component. Before presenting the formal transformation, we describe informally the underlying idea.

Service locking is implemented by a selection connector that intercepts all incoming trigger activations, and forwards them only under conditions that correspond to an unlocked service. To account for the port group synchronization, we add a dummy component for each input and output group (i.e., a primitive component that simply forwards data without modifying it). These components also serve to control the service locking selection connector, by signalling events corresponding to the flattened composite being entered and exited. Naturally, these additional model entities contribute to the complexity of the flattened model. After presenting the basic transformation, we will show how this overhead can be reduced.

Figure 4 illustrates the added constructs when flattening is applied to an instance of the component depicted in Figure 3.

In the description of the flattening transformation, the trigger port of a port group (input or output) G is denoted

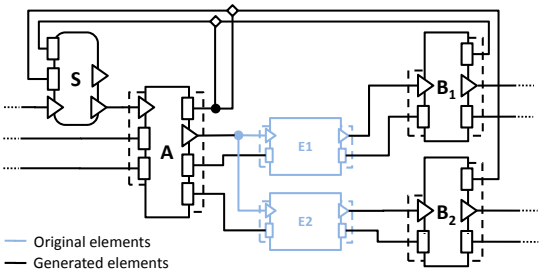

Figure 4. Result of the flattening transformation. The circle and diamonds are shorthand notation for *fork-* and *or* connectors, respectively.

G^{t}, and G^{d} denotes the data ports of G. To simplify the presentation, we denote by G^{d}_n the n^{th} data port of G.

Starting at the component selected for flattening, the transformation is applied recursively to all subcomponent instances down to instances that are either primitive or marked as *partitioning points*. To flatten an instance of a composite component, we first replace it by its constituent elements. Then, for each service of the composite, with I denoting the input port group and $O = \{O_1, \ldots, O_{|O|}\}$ the output port groups of the service, we perform the following steps:

1) Add a selection connector S, with boolean data ports $s_1, \ldots, s_{|O|}$ initialized to *true*, one trigger input port s and two trigger output ports sl and su (representing the service being *locked* and *unlocked*, respectively). The selection criteria of S are:

$$su: \; s_1 \wedge \ldots \wedge s_{|O|} \qquad sl: \; \neg(s_1 \wedge \ldots \wedge s_{|O|})$$

2) Add a new component A, with one input group AI identical to I, and one output group AO with $AO^{\mathrm{d}} = I^{\mathrm{d}} \cup \{a\}$, where a is a data port of type boolean. The behavior of A is to simply copy data from input to output ports, and to write *false* to port a.
3) Add a *data-fork* connector F, with one input port f and output ports $f_1 \ldots f_{|O|}$.
4) For each $m \in \{1 \ldots |O|\}$ add a *data-or* connector C_m, with two input trigger ports cu_m and cl_m, and an output trigger port c_m.
5) For each $m \in \{1 \ldots |O|\}$ add a component B_m, with one input group BI_m identical to O_m, and one output group BO_m with $BO^{\mathrm{d}}_m = O^{\mathrm{d}}_m \cup \{b_m\}$ where b_m is a boolean data port. The behavior of B_m is to copy data from input to output ports and to write *true* to port b_m.
6) For each $m \in \{1, \ldots, |O|\}$, add the following connections:

$$\langle su, AI^{\mathrm{t}} \rangle \quad \langle a, f \rangle \quad \langle f_m, cl_m \rangle \quad \langle c_m, s_m \rangle \quad \langle b_m, cu_m \rangle$$

7) For any connection matching the left-hand side of an expression below, replace it by the right-hand side

connection (where p and p' are ports outside and inside the composite, respectively, and $n \in \{1, \ldots, |I^{\mathrm{d}}|\}$):

$$\langle p, I^{\mathrm{t}} \rangle \Rightarrow \langle p, s \rangle \qquad \langle I^{\mathrm{t}}, p' \rangle \Rightarrow \langle AO^{\mathrm{t}}, p' \rangle$$
$$\langle p, I^{\mathrm{d}}_n \rangle \Rightarrow \langle p, AI^{\mathrm{d}}_n \rangle \qquad \langle I^{\mathrm{d}}_n, p' \rangle \Rightarrow \langle AO^{\mathrm{d}}_n, p' \rangle$$

8) Similarly, replace connections matching the left-hand side of an expression below (where $m \in \{1, \ldots, |O|\}$ and $n \in \{1, \ldots, |{O_m}^d|\}$):

$$\langle p', {O_m}^{\mathrm{t}} \rangle \Rightarrow \langle p', {BI_m}^{\mathrm{t}} \rangle \qquad \langle {O_m}^{\mathrm{t}}, p \rangle \Rightarrow \langle {BO_m}^{\mathrm{t}}, p \rangle$$
$$\langle p', O^{\mathrm{d}}_{mn} \rangle \Rightarrow \langle p', BI^{\mathrm{d}}_{mn} \rangle \qquad \langle O^{\mathrm{d}}_{mn}, p \rangle \Rightarrow \langle BO^{\mathrm{d}}_{mn}, p \rangle$$

C. Optimization

The flattening transformation defined above is developed to guarantee operational equivalence in all possible scenarios, by adding constructs implementing key behaviours of the removed composite. Next, we will identify situations where the transformation can be simplified without compromising the operational correctness.

Service lock optimization: Let E be the composite component to flatten, and t the input trigger port of one of its services. If there exists a composite component E' such that E is a subcomponent of E' (on any level of nesting), and t can be reached only from one input trigger port t' of E', and only once per activation of t', then the service which t belongs to does not require locking, since this is already ensured by the locking of the enclosing service.

Concretely, this means that the *selection* S, the *data-fork* F, the *data-or* connectors $C_1 \ldots C_{|O|}$ and the associated connections are not needed for this service. Instead, the connection from an external trigger port leads to the trigger port of A instead of S, i.e., connections $\langle p, s \rangle$ and $\langle su, AI^{\mathrm{t}} \rangle$ are replaced by the connection $\langle p, AI^{\mathrm{t}} \rangle$. Also, since the ports a and b_m are now unconnected, they can be removed from A and B_m, respectively. Figure 5 illustrates this optimization.

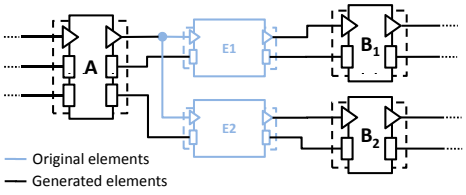

Figure 5. Result when service lock optimization is possible.

Group synchronization optimization: An input- or output port group G of the flattened composite component can be ignored if

i) all groups that trigger G (directly or through connectors) also write to all data ports of G; or

ii) no group that is triggered by G (directly or through connectors) receive data from any other group than G.

Under these conditions, the synchronization performed at group G is also performed when data is produced (case *i*) or received (case *ii*).

Ignoring an input group (I in the definition above) means that the connections from the external data port leads directly to ports of internal components or connectors instead of leading to A. Formally, $\langle p, AI^{\mathrm{d}}_n \rangle$ and $\langle AO^{\mathrm{d}}_n, p' \rangle$ are replaced by $\langle p, p' \rangle$ for all $n \in \{1, \ldots, |I^{\mathrm{d}}|\}$. Moreover, these (now unconnected) input and output data ports can be removed from A.

Similarly, the result of ignoring an output group (O_m in the definition) is that connections $\langle p', BI^{\mathrm{d}}_{mn} \rangle$ and $\langle BO^{\mathrm{d}}_{mn}, p \rangle$ are replaced by $\langle p', p \rangle$, for all $n \in \{1, \ldots, |{O_m}^d|\}$. One example of this optimization is illustrated in Figure 6.

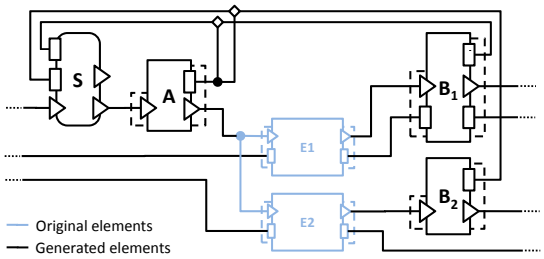

Figure 6. Result when group synchronization optimization is possible for the input group I and for the second output group O_2.

Combining the two optimizations: In the case when an input group I can be ignored and the service locking optimization has been applied, then A can be completely removed, and the connections $\langle p, AI^{\mathrm{t}} \rangle$ and $\langle AO^{\mathrm{t}}, p' \rangle$ are replaced by $\langle p, p' \rangle$.

Similarly, when an output group O_m can be ignored and service locking optimization has been applied, then B_m can be completely removed, and the connections $\langle p', BI^{\mathrm{t}} \rangle$ and $\langle BO^{\mathrm{t}}, p \rangle$ are replaced by $\langle p', p \rangle$.

D. EFP of the Generated Elements

Regarding the assignment of EFP values to generated elements, in the case of our ProCom transformation it is primarily the introduced primitive components that require EFPs to be analyzable. For this, we extend the ProCom attribute framework [2] by allowing value computation functions (represented by QVT query expressions) to be associated with each EFP type. The functions define, for that particular EFP type, how the value should be computed for A and for the B_m components.

Considering the WCET analysis presented in [3], the execution time of all generated components consists of the time required to copy data from input to output ports, which can be computed from the number of ports and their types. The execution time of the other constructs do not need to be explicitly computed and represented in for of an attribute, since the overhead of connectors is taken into account by the analysis.

Table I
EXPERIMENT RESULTS.

Source Model				Target Model		Transform. time
hd	**bf**	**Inst.**	**Conn.**	**Inst.**	**Conn.**	
2	5	30	24	35	39	297 ms
2	6	42	28	48	46	375 ms
2	7	56	32	63	53	484 ms
3	5	155	124	185	214	829 ms
3	6	258	172	300	298	1 344 ms
3	7	399	228	455	396	2 253 ms
4	5	780	624	935	1089	3 391 ms
4	6	1554	1036	1812	1810	7 016 ms
4	7	2800	1600	3199	2797	74 563 ms

IV. VALIDATION

A formal proof that the flattening transformation does preserve all aspects of the operational semantics is outside the scope of this paper. However, the fact that the semantics has been formally defined [4], provides a good starting point for this part of the validation. Constructing the timed automata defining the semantics of a composite component and of the corresponding flattened structures, respectively, is straightforward, but ensuring that they are indeed equivalent is not quite as simple and will be addressed in our future work.

Instead, as a first step of validation, we show that the proposed transformation is not prohibitively costly. The flattening transformation has been straightforwardly implemented as an operational QVT transformation, and applied to a number of system models of varying complexity. The results are summarized in Table I.

For given parameters *hd* and *bf* (representing *hierarchical depth* and *branching factor*), a source model is created of the given depth, and where each composite component contains *bf* subcomponent instances and four connectors. The total number of component instances and connectors in the source model is thus exponential with respect to the hierarchical depth.

As expected, the resulting models are larger – as a result of the introduced elements – but the difference is not that big. The reported execution time for the transformation is the average of ten transformations, performed on a dual core CPU with 2,79 GHz and a memory space of 3,48 GB of RAM. We observe that even for big systems containing a thousand component instances, the flattening process takes only a couple of seconds.

Although scalability of the transformation to handle extremely large models is not our primary concern, the high execution time in the last experiment raises some concerns. There are no conceptual reasons for this, and it is likely to be an effect of linear searches in QVT, e.g., searching through the list of connections to find the one leading out from a particular port, since the model contains no explicit reference from port to connection, only in the opposite direction. As stated in [5], existing QVT engines are partial implementations with room for improvement. Thus, we believe that scalability can be significantly improved by a more careful transformation implementation, but further experiments would be required to investigate this.

The two optimizations discussion in Section III are not included in the experiments, since their impact depends to a large degree on the way components are connected. The experiments needed to yield conclusive results, by automatically generating composites with different kinds of representative connection patterns, are part of our planned future work.

V. RELATED WORK

From the categories described by Woodside et al. [6], our work focuses on mixing early analysis which rely on the component model and measurement based analysis used to compute properties of primitive components and providing efficient way to deal with analysis scalability issues which remain a main concern. Our approach is not specific to performance analysis but can be applied for other purposes than performance ones.

Some works have proposed operational semantic preserving analysis to enable and/or simplify analysis and code generation.

Lasnier et al. [7] propose an incremental transformation which refines the considered AADL model into a flattened model for analysis and code generation purposes. Comparing to our work, the approach is AADL specific, the number of considered nested levels is small and the AADL models do not have well-defined formal semantics.

Bozga et al. [8] describe model transformation techniques in the scope of the BIP component model. These model transformations include flattening and optimization steps similarly to the model transformation presented here. The main differences are *(i)* in the flexibility of the model transformation – i.e. the possibility to reuse existing components without altering their structure – and *(ii)* in the semantics of the component model. As a consequence of this semantic difference, the considered optimization steps are very different.

Asztalos et al. [9] propose a framework for formal verification of model transformations, by checking pre and post conditions over transformation rules. The technique is applied to a model transformation that aims at flattening an input model. It would be interesting to consider this kind of results to verify our model transformation process in order to ensure correctness of our flattening transformation is correct. Verifying the optimization transformation would be more difficult since it requires a deeper interpretation of the component model semantics.

In the work of Lublinerman et al. [10], the hierarchy of components is based on synchronous block diagrams.

As a consequence, the consistency of a model can only be assessed once the complete hierarchy of components have been flattened, or using a specific profiling technique (presented in [10]) which delays the consistency checking until the complete graph of profiles is available. Considering another component semantics gives more flexibility in the flattening process.

A generic tool chain for the Think component model is proposed by Leclercq et al. [11]. Models of individual components are merged and explicit links are derived from naming convention and reside in memory during compilation process. Contrasting with our approach which aims to simplify the component model used by analysis, it allows plugging of additional compilation steps which enrich the component model with additional information such as extra functional properties.

VI. Conclusion

We have described a general approach to allow compositional and non-compositional analysis techniques to be used on hierarchical component models, by means of a partial flattening transformation that preserves key semantic aspects. In the case of compositional analysis, the proposed approach facilitates tradeoffs between scalability, precision and reuse concerns. The presented work improves management of extra-functional properties by simplifying integration of existing analysis techniques.

The general approach has been exemplified in the context of ProCom, a component model specifically targeting real-time embedded systems, by presenting a flattening transformation guaranteed to preserve the operational semantics of ProCom composite components also in the flattened model. Investigating the applicability of the general method to other component models will be part of our future works.

In this paper, flattening has been presented in the context of analysis, but the approach is also applicable as a first step of code synthesis. Then, the possibility of partial flattening means that some components in the system hierarchy can be synthesized specifically to be reusable in any context, while others can be aggressively optimized for the particular context in this system.

Using the same flattening mechanism for analysis and synthesis also helps reducing the gap between model-level analysis and the properties of the generated code. For example, a difficult part of the approach is to define EFP of the constructs that are introduced by the transformation, and knowing that the same constructs will be considered during synthesis can significantly simplify this tasks.

Our future works include establishing a formal proof of operational semantic preservation, implementation of model transformation variants which preserves different properties, investigating the role of partial flattening for code synthesis.

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] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, and I. Crnković, "A component model for control-intensive distributed embedded systems," in *11th Int. Symposium on Component Based Software Engineering*. Springer, 2008.

[2] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković, "Integration of Extra-Functional Properties in Component Models," in *12th International Symposium on Component Based Software Engineering*. Springer, 2009.

[3] T. Leveque, E. Borde, A. Marref, and J. Carlson, "Hierarchical composition of parametric WCET in a component based approach," in *14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*. IEEE, March 2011.

[4] 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*, 2009.

[5] I. Kurtev, "State of the art of QVT: A model transformation language standard," in *Applications of Graph Transformations with Industrial Relevance*, ser. Lecture Notes in Computer Science. Springer Berlin, 2008, vol. 5088, pp. 377–393.

[6] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 171–187.

[7] G. Lasnier, L. Pautet, and J. Hugues, "A model-based transformation process to validate and implement high-integrity systems," in *14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing*. IEEE, March 2011.

[8] M. Bozga, M. Jaber, and J. Sifakis, "Source-to-source architecture transformation for performance optimization in BIP," in *IEEE International Symposium on Industrial Embedded Systems*, 2009, pp. 152–160.

[9] M. Asztalos, L. Lengyel, and T. Levendovszky, "Towards automated, formal verification of model transformations," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2010, pp. 15–24.

[10] R. Lublinerman, C. Szegedy, and S. Tripakis, "Modular code generation from synchronous block diagrams: Modularity vs. code size," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2009, pp. 78–89.

[11] M. Leclercq, A. E. Özcan, V. Quéma, and J.-B. Stefani, "Supporting heterogeneous architecture descriptions in an extensible toolset," in *29th International Conference on Software Engineering*. IEEE Computer Society, 2007.