Johan Ersfolk | Johan Lilius
Torbjörn Lundvist | Ivan Porres

A Tool for Efficient Combination and Evaluation of Reusable Design Assets

Turku Centre *for* Computer Science

TUCS Technical Report
No 975, April 2010

A Tool for Efficient Combination and Evaluation of Reusable Design Assets

Johan Ersfolk
Johan Lilius
Torbjörn Lundvist
Ivan Porres

TUCS Turku Centre for Computer Science
Department of Information Technology
Åbo Akademi University
Joukahaisenkatu 3-5A, FIN-20520 Turku, Finland
johan.ersfolk@abo.fi, johan.lilius@abo.fi,
torbjorn.lundkvist@abo.fi, ivan.porres@abo.fi

TUCS Technical Report
No 975, April 2010

Abstract

Research on embedded system design typically focuses on design space exploration in the architecture platform space and the goal is to obtain an optimal implementation of the system. In the mobile phone industry the design problem is often quite different. The goal is not to design a new system but to add a use case to an existing product or to a family of products. In this case it is important to be able to quickly find possible performance problems caused by the simultaneous use of the new use case in conjunction with existing use cases. In this article we propose a structure for the design flow and show how a dedicated tools is constructed using a model based approach.

Keywords: Model Based Development, design space exploration, component reuse, domain-specific modeling

1 Introduction

Modern embedded systems such as mobile phones perform tasks similar to those of personal computers. Tasks such as video playback, text processing, web browsing and running complex user interfaces has become standard features of hand held devices, at the same time, mobile phones are required to run real time tasks such as GSM. While all the new applications requires more processing power, hand held devices run most of the time on battery power which puts constraints on the energy consumption of the system and thereby also on the processing capacity. To design such a system without breaking any performance requirements on a platform with limited resources is a difficult task.

While existing embedded system design methodologies focus on design space exploration in the architecture platform space, the situation in the mobile phone industry is usually that there is a number of fixed platforms for which new applications are being developed using libraries of existing software components. A typical scenario would be that a set of new *use cases* needs to be implemented for an existing system, this would involve for example adding video playback (a use case) and video recording capabilities (a second use case) to a phone. The design problem is then to find the minimal changes to the existing architecture to implement the new use cases, we therefore need to evaluate how the new *use cases* will perform on the given platform and how these can co-exist with existing *use cases*.

The design flow depicted in figure 1 highlights the communication between a system architect and the teams working on the different subsystems. To begin the design the system architect decomposes the new use cases into subsystems. For the existing system the subsystems are available as *assets* in a library, from which relevant performance data can be obtained. For the required new subsystems, the system architect requests estimates from the designer team responsible for the technical subsystem (e.g. for the video encoding from the media subsystem team). Using these estimates the system architect can start evaluating the system model for its performance. The obtained values are given back to the designers as a *budget*, to use in the implementation of the subsystem. The verification of the implementation will give some feedback to the system architect, which may lead him to make changes to the design. Once the component is deemed ready it is integrated into the final product. At this stage the component becomes an *asset*, which means that the component has been deployed in a finished product, and proved to work in conjunction with the other components in the use cases.

Since the system architect has an overall responsibility over the design process, he needs powerful tools, based on dedicated analysis models, to support him in his design work. In this article we approach this issue with a model driven approach using our metamodeling tool Coral [11]. In section 3 we present the requirements for the design methodology and the EFCO tool which has been implemented to demonstrate the design flow. In section 5 we describe how Coral has been used to construct the tool. This article is partially based on our previous

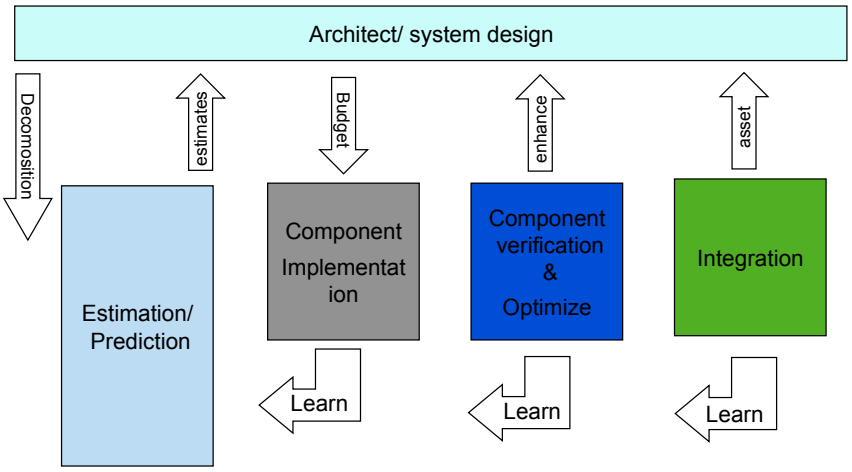

Figure 1: A design flow and its feedback loops

work [6] and extends this work by describing the tool design methods more in detail.

2 Background and Related Work

The presented design flow has several similarities with the Y-chart approach [9]. The Y-chart approach separates the application models from the architecture instance models. A set of architecture instance models can be evaluated against a set of application models and the models can be reused separately in other projects. For a given architecture instance a performance model needs to be created. Performance analysis for a specific architecture instance can then be done after the set of applications have been mapped to the architecture instance. The results from this performance analysis can be used to make improvements on the architecture instance, the applications themselves or on the mapping between application and architecture instance. This process can then be repeated until an architecture that satisfies all constraints is found. In our approach, the separation between a platform independent and a platform dependent model does not properly exist as the goal is that all the assets used in the models contain as much platform dependent information as possible. The reason for this is that the performance analysis of a system becomes more precise when more of the components contain accurate performance information. The reason why this is possible is because the hardware dependent values are going to remain stable throughout the life-time of a product family, or their changes can be predicted through discussions with the silicon vendors.

The tool presented in this article is implemented as a front-end to CoFluent Studio [1, 4]. Cofluent Studio is an embedded system design tool that enables performance analysis of hardware/software systems, by using the Y-chart approach.

The behavior of components or whole systems can be designed in CoFluent Studio using a combination of a graphical notations and C code. The EFCO tool creates libraries from the components imported from CoFluent Studio, which are used in the design flow for constructing the simulation models; the generated models are then exported back to CoFluent Studio which generates a SystemC test bench for performance analysis. The EFCO tool extends CoFluent Studio by adding support for efficiently re-using and combining *use cases* and *features*.

The Architecture Analysis and Design Language (AADL) [7] is used to model the software and hardware architecture of embedded real-time systems. It contains constructs for modeling both software and hardware components and is used for analysis such as schedulability and flow control. Compared to our approach, our models could be exported to AADL and be used for analysis instead of the SystemC simulator generated by CoFluent Studio.

3 Design Space Exploration using the EFCO tool

The EFCO tool has been designed to support the system architect in the described design flow. It provides a Feature library tool for efficient management and reuse of library elements, it has also been designed as a Use Case evaluation tool that combines common functionality of the use cases and generates a simulation model of the system. The generated simulation model allows the architect to simulate different combinations of the available *use cases* and to analyze these for potential performance bottlenecks. The EFCO tool is implemented as a front end tool to CoFluent Studio [1].

3.1 Use Case Based Evaluation

A mobile phone supports a large number of use cases, for example video playback, file download and GSM call. When adding new use cases to the system, the design will consist of an old system structured as a set of use cases and a set of new uses cases. Then the first question to be answered is whether the new use cases can be run on the given platform. This is standard fare. However, the challenge comes when there is a need to support new use cases concurrently with the old ones. The performance of a single use case directly depends on which other use cases are active concurrently. Because of this it is important to consider which use cases can be active simultaneously and to evaluate how these will compete for resources. As the number of possible use case combinations on a system can be large it is important to have efficient tools to analyze use case combinations with different parameters rapidly.

Use Case Combination. When new use cases are added to the model of an existing system the use cases will be mapped to the available set of resources. In practice this means that we can use the new model to analyze how the system

performs when more tasks are mapped to the available resources of the platform. What is more important is that the impact of features being shared between use cases can be analyzed, this is needed as some features provided by a system are such that the functionality is not duplicated but shared. Functionality which can be tied to a resource is for example communication channels, accelerators, peripheral devices etc. When generating the new system model, the shared components must be combined so that such functionality is shared and not duplicated. As an example, we would not create a new network device for every application that uses TCP, instead the features describing the communication functionality will be shared by the use cases.

A *use case* describes one functionality of the system, such as video playback, on an abstract level; in order to specify the different alternatives how to accomplish the task the *use case* can consist of several *use case modes* that describe a particular instance of the *use case*. In case of the video playback *use case*, possible *use case modes* could be 1) video playback over TCP or 2) video playback from local storage. For the use case combinations this means that even if two use cases share a feature, the features needed by a use case depends on which mode is active.

Combining functionality of the use cases implies that we need to transform the model, in order to make the design work efficient it is important to have good tool support. For this purpose we have developed a simple graph merging algorithm [8] that given two *use case modes* (or features) creates a new model, which contains the behavior of both *use case modes*. The new model will contain shared elements and can therefore be analyzed for problems in resource contention.

To support re-use we need to store designs and performance information in a library. The system should also keep information about existing designs, in the case a whole system is reused such information as performance and mapping would not change and only the new use cases are added to the system. The EFCO tool supports this by providing specific projects for library components, when importing a project from CoFluent Studio a library project it is placed in such a library. The library components can be re-used on different levels, whole use cases can be re-used but also single features and services.

Flow Control. When performance problems caused by shared resources are found in the model, it is a huge benefit if it can be solved without changing the platform. If the shared resource does not have enough capacity for the *use cases* the architecture might need to be modified or the *use cases* cannot be allowed to execute concurrently. However, most often the problem of sharing a resource is due to stochastic behavior of data streams and badly tuned mechanisms for handling the resource contention. In order to resolve resource conflicts and find the optimal parameters for the system the system architect can use different flow control mechanisms.

The control mechanisms we have considered are based on feedback and control tasks with respect to the data flow. Such protocols as window flow control

and XON/XOFF are well known in computer communication; in general these control the number of events waiting either as the backlog of a receiver or the total backlog of the elements in the control loop. By using such constructs the *use cases* can be forced to share resources depending on how critical the tasks is and also depending on the current state. The other alternative is to give the operating system scheduler the task based on priorities.

Considering a data stream, over a network or a DMA transfer, the behavior is often bursty with jitter. A use case depending on such a stream will also inherit the behavior of the input stream as it will have to wait for data between burst; as also resources on the system, such as processors or communication channels, are shared the behavior of the use cases gets even worse. If the use case for example is a video decoder, a bursty output stream might lead to a situation when the playout buffer underflows. To prevent a use case from adding unwanted behavior to the system these is a need to control some tasks depending on some criteria. An example including flow control is included in section 4.

In the EFCO tool, flow control can be added to any pair of features. The flow control consists of two features as one of the features is controlled and the other one is observed. When adding flow control to a feature the feature gets a special data source/sink for the feedback messages. The feedback messages can travel on data channels, if the data path is shared and consists of features/channels common to other parts of the system these can be combined. The flow control tool provides a simple tool to measure how the use cases needs to be controlled to enable different use case combinations to run on the platform.

Simulation Use Case. When the model is ready to be analyzed it is transformed to include the extra details needed for the simulation. This transformation adds information, such as routers and additional parameters, to the model that was not needed in the single use cases but is needed as a result of the use case combination.

We need routing in two different situations, 1) for sending data through the chosen use case mode, e.g. transmit over WLAN or GSM and 2) for deciding to which use case a message belongs, e.g. if the message from the WLAN feature belongs to the video decoder or audio decoder use case. The routing is accomplished by adding a header to the messages and adding routers to the design. The routers analyze the headers and sends the messages in the correct direction without modifying the messages or adding delay. The routers abstract away the implementation of routing messages through the use case as the architect is only interested in analyzing the performance. The simulation use case includes all the information needed to export the model to CoFluent Studio for simulation.

3.2 The EFCO Metamodel

The modeling language used in the EFCO tool is constructed by a set of structuring concepts encoded into a metamodel. The metamodel can be split into 3 parts.

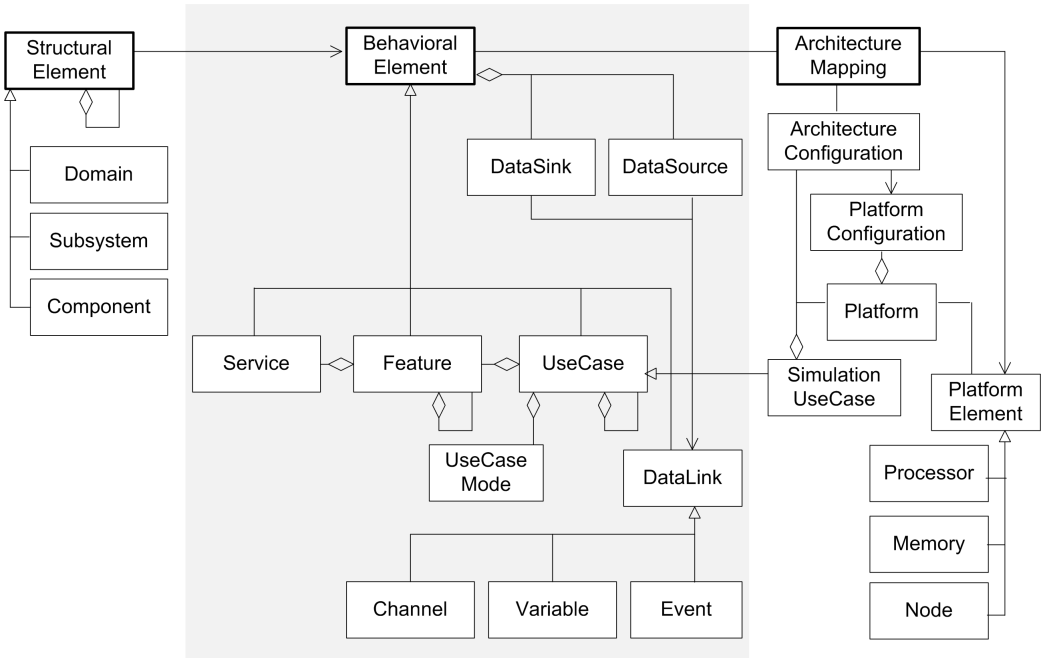

Figure 2: The structureal metamodel (left), the behavioral metamodel (middle) and the platform metamodel (right). The metamodels have been simplified in order to increase readability.

The **Structural Model** is used to delineate parts of the system according to particular responsibilities (e.g. the multimedia subsystem), the **Behavioral Model** is used to describe the functionality of the system and the **Platform Model** describes the hardware architecture of the system. These can also be seen as a hierarchy, where the structural model is at the highest level and the platform model at the bottom, but note that a structural relationship of inclusion does not necessarily imply a corresponding relationship on the platform level, since the same subsystem can be mapped onto different platform elements. In this article we will concentrate on describing the behavioral model as it is central to our modeling approach. More detailed information about the models can be found in [8, 12].

Behavioral model. The behavioral metamodel (c.f. Figure 2) supports a use case driven decomposition approach. The decomposition has as a starting point a *use case*. A *use case* describes a general functional scenario of a system; it looks at the system from the point of view of the end-user. Thus a use case is very generic like "video playback". Typically this is too generic and the *use case* has to be refined into *use case modes*. A *use case mode* describes a specific way the use case will be implemented and thereby fixes the communication network topology of the *use case*. For the video playback *use case* we could have 3 modes: video playback from internal memory, over 3G or over Wireless LAN. Typically a phone might support internal memory and 3G, and the Wireless streaming would be an option for a more high-end phone, depending on which *Features* the radio

subsystem contains.

The *use case mode* is decomposed into *features* and *services*. *Services* act as the elements at the lowest level of granularity and are used to compose *features*. A *feature* can then be composed of other *features* and *services*. A *feature* describes a more general functionality that can be reused in other *features* or *use cases*. A *feature* can be composed of a number of *services* and/or other *features*. *Features* correspond to functions with a structure and no behavior of their own in CoFluent Studio's functional model (example: an MPEG decoder). A *service* describes a specific functionality that can be reused and combined in different *features* or *use cases*. A *service* is the element with the lowest granularity in a *use case* and they can be said to be atomic, since they cannot be split up. *Services* correspond to functions with a behavior in CoFluent Studio's functional model (example: a DCT function in an MPEG decoder).

Although we use the term behavioral to characterize the part of the meta-model used to structure the functionality of the design, we do not propose a new approach to describe the functionality of atomic elements (services), but rely on the approach of the underlying simulation tool for this (e.g. the Timed-Functional approach of CoFluent studio is used in our tool).

4 The EFCO Tool Workflow: Examples

In this section we will give an example of the workflow supported by the EFCO tool. The example will show the steps a system architect takes to design a system based on an existing system.

Step 1. The existing model and new use cases. The existing system consists of a number of use cases, for simplicity we will only consider GSM call, call over TCP and file download. The new use case to be implemented is a video decoder with the three modes: playback over 3G or LAN and playback from a local storage. To model the behavior of the decoder the decoder is implemented in CoFluent Studio and is then imported to the EFCO tool. When modeling the decoder in CoFluent the designer also created a video source and sink, this use case can be imported to the EFCO tool either completely as a use case or as features. In this example we need to add 3G and LAN features to the use case.

The excisting system, including the use cases and the mapping of these to a platform, is available as an EFCO library and is imported to the tool. The network features needed by the decoder are also imported to the decoder use case, these features are already in use in the existing system as it includes functionality such as file download. The use cases are illustrated in figure 4, figure 5 shows the imported video decoder including the reused network features and figure 3 shows the existing system.

Step 2. Combination and simulation. As can be seen in figures 3 and 5, the use cases share some features, these features should be combined when we

create the simulation model describing the whole system. Combining these *use cases* we obtain a new model which is shown in figure 6, this model describes the whole system and is used to evaluate the system including the new *use cases*. The new model is transformed into a *simulation use case* before it can be exported to CoFluent Studio for simulation. The *simulation use case* includes router for routing messages through the model and parameters for choosing which *use cases* are active and in which mode the *use cases* operate. The *simulation use case* is the exported to CoFluent Studio for evaluation.

Step 3. Evaluation and flow control. During the simulation possible performance problems can be found. If we consider the *use case combination* of the audio and video decoders, we notice that input streams to the decoders are received over a network and are therefore bursty. To prevent that the playout buffers of the audio and video decoders underflow, the system architect can add feedback (dashed lines in figure 6) from the playout buffers to the decoders. If either decoder is late, i.e. the level of the playout buffer is low, it will get more processor time than the other tasks. The parameters that enable the *use cases* to co-exist are used when the system is implemented.

5 Tool Implementation

The EFCO tool allows the architect to easily simulate different applications executing concurrently on a number of architecture platforms and to analyze the resource sharing between the applications. The tool combines common functionality of the use cases and creates a model for analyzing resource contention, if needed the architect can further tune the performance of the use cases by adding flow control to critical parts of the design. In order to make the evaluation efficient the tool allows re-use at different levels of abstraction, when whole systems are reused it is important to keep platform mapping and performance data from the previous designs, thereby the architect only needs to focus on the new use cases.

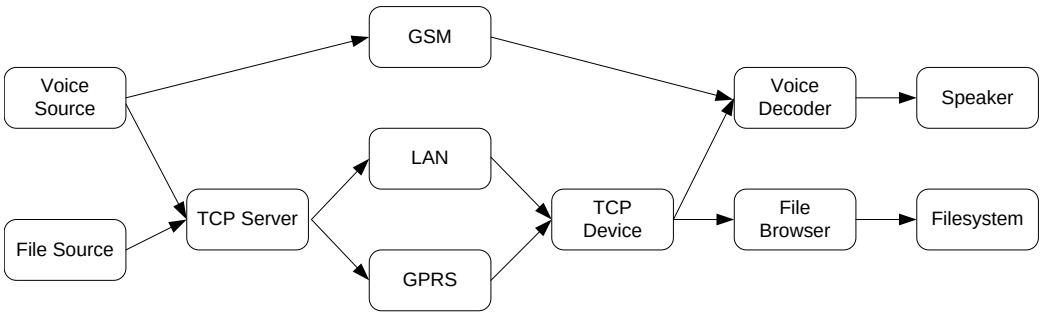

Figure 3: The existing use cases obtained from an EFCO library.

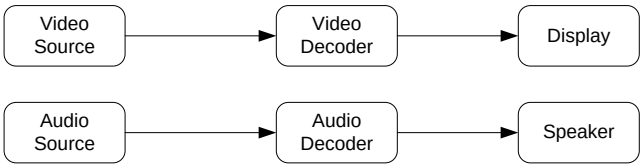

Figure 4: The new use cases, imported from CoFluent Studio.

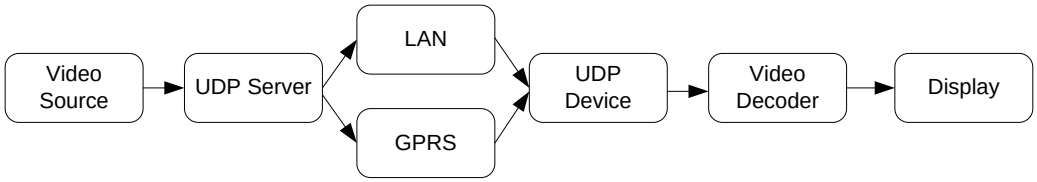

Figure 5: The video decoder use case including re-used features.

In this section we will present the EFCO tool, which provides a design environment for EFCO projects. The EFCO tool consists of the EFCO modeling language, a set of editors and viewers for the EFCO models and diagrams, and integration tools for CoFluent Studio. We will also present how the EFCO tool was implemented using the Coral Modelling Framework.

5.1 Modeling Framework Principle

The EFCO tool extends the functionality of CoFluent Studio. However, as CoFluent Studio is not an open tool allowing customization, the EFCO Tool is implemented as an extension of the Coral Modeling Framework [11]. The Coral Modeling Framework is based on the Object Management Group (OMG) [13] standards and provides framework to design and construct new modeling languages and editors. Coral is also a highly customizable modeling tool, it can be extended with new modeling languages and editors to provide customized design environments for creating, editing and transforming models in these languages. Coral is a *metamodel independent* tool, this means all modeling languages and models are treated uniformly by the tool. That is, all modeling languages are defined using a common metamodeling language [2], similar as the OMG UML 2.0 Infrastructure [14]. Coral has a set of built-in editor components, such as the diagram editor and property editor, which are metamodel independent and configured to support a particular modeling language. Coral is designed on the important principle that providing editor support for a new modeling language does not require programming using a tool-specific API. Instead editor artifacts are defined declaratively using a set of configuration models. Coral can also be used for running model transformation programs, based on either rule-based double-pushout transformations or a Python [16] programming interface.

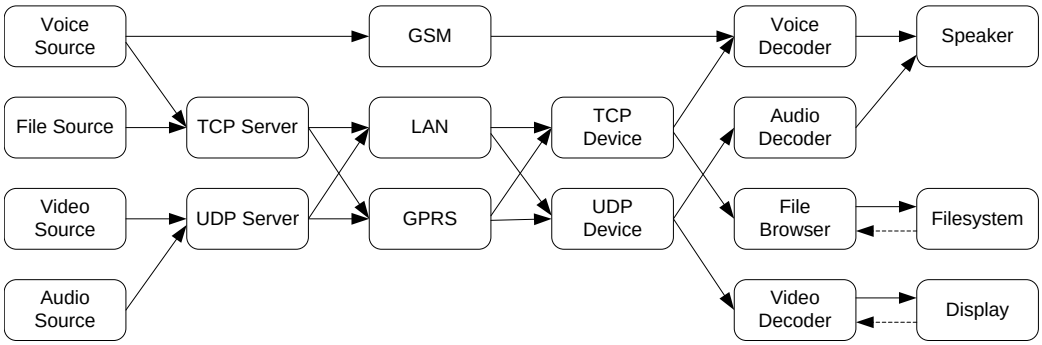

Figure 6: The new combined use case, the common features has been combined.

The EFCO tool supports the graphical editing of models using several diagram editors. Diagram editors in the EFCO Tool are constructed using the tools provided by the Coral Modeling Framework. In Coral, the diagrams or the concrete syntax and the models or the abstract syntax are defined and maintained as two different but related artifacts. That is, the abstract syntax is defined using a metamodeling language and the concrete syntax is defined using a diagramming language.

Diagram support in Coral is based on the UML 2.0 Diagram Interchange (DI) standard [15]. This standard is used by the tool both internally for maintaining the structure and contents of the diagrams as well as externally for interchanging models with diagrams. Since diagrams and models are separate artifacts it enables the use of several diagrams for the same abstract model data as well as using different graphical notations for the same model elements depending on the context. In Coral, the concrete syntax is defined using a set of mappings between the abstract and concrete syntax, with a special mapping language called the Diagram Interchange Mapping Language (DIML) [3]. Using this language, each model element is mapped to a parameterized skeleton of diagram elements. The DIML mappings are then used by a *diagram reconciliation component*. The diagram reconciliation component evaluates these mappings in the context of the abstract model to create new diagrams. This component is also used incrementally, to propagate changes in the abstract model to the diagrams, keeping diagrams up to date. This has the important advantage that diagram updates are completely decoupled from changes in the abstract models, allowing new editors to be constructed to deal with the abstract syntax alone. Hence, diagrams are kept up to date regardless of which editor component or model transformation engine changed the model.

It must, however, be noted the output from the diagram reconciliation is a diagram according to the DI standard, which describes only the structure of the diagram. In order to view the diagram it must be rendered to the language specific notation. In Coral, visual notations are defined declaratively using a language based on *parametrized Scalable Vector Graphics* (SVG) [17]. Each diagram el-

ement can be associated with many symbols, of which one is selected based on the context of the diagram element or the underlying abstract model. The final notation is achieved by applying these definitions in the context of the model and the DI diagram. This mechanism is fully integrated with the diagram viewers and is used to draw diagrams both on screen and to files.

5.2 Graphical Editors

As discussed in the previous section, the EFCO Tool extends the functionality of CoFluent Studio by introducing concepts of combining reusable components, such as Use Cases, Features and Services, and maintaining a library of reusable components. The actual design of the individual components is carried out using CoFluent Studio. Therefore it is important that the EFCO Tool is not seen as a replacement for the editors provided by CoFluent Studio.

The EFCO Tool supports importing components from CoFluent Studio projects into a library. CoFluent uses a tool-specific XML-based syntax to store data, whereas the Coral Modeling Framework used to build the EFCO tool uses models represented as graphs. The CoFluent projects are converted into an EFCO library using a conversion tool executed when a CoFluent project is loaded into the EFCO. In this library, the components to be reused are stored as *library elements* in a separate library project from which they can be instantiated for reuse in new EFCO projects.

New EFCO projects can be designed graphically using diagram editors. Perhaps the most important diagrams used in the EFCO Tool are the Use Case diagrams, which are used to design new functionality of the system by connecting Features, and the Feature diagrams, which in turn shows how a Feature is composed from services.

The Use Case diagram editor supports the creation of new Use Cases using two methods: by starting from an empty Use Case or by using an existing Use Case from a library as a starting point. In the case of an empty Use Case, a new model element and a model element is simply created. When existing Use Cases from a library are used, a copy of the library Use Case is first inserted into the project by a model transformation program, after which a new Use Case diagram is created by the diagram reconciliation component. This is to ensure that the library remains intact. To these Use Case diagrams new Features can be added by dragging the library element from the library to the diagram. Similarly, the Features including their dependencies are inserted as a copies into the Use Case by a model transformation program. The diagram editor has a set of toolbar buttons. Each of these execute a model transformation action defined by a set of declarative double pushout transformation rules, consisting of a left hand side (LHS) and right hand side (RHS). These rules are then executed by a model transformation engine which applies these rules on a model. These rules can be arbitrarily defined, however, in the EFCO diagram editors, these are are used to insert new elements into

the diagram, for example connections between channels in a Feature, sources or sinks, parameters or actors. In addition to the toolbars, the diagram editors also supports moving connections between channels using free-form editing gestures. In Figure 7 a screenshot of the EFCO Tool is shown. In the figure a Use Case diagram is being edited.

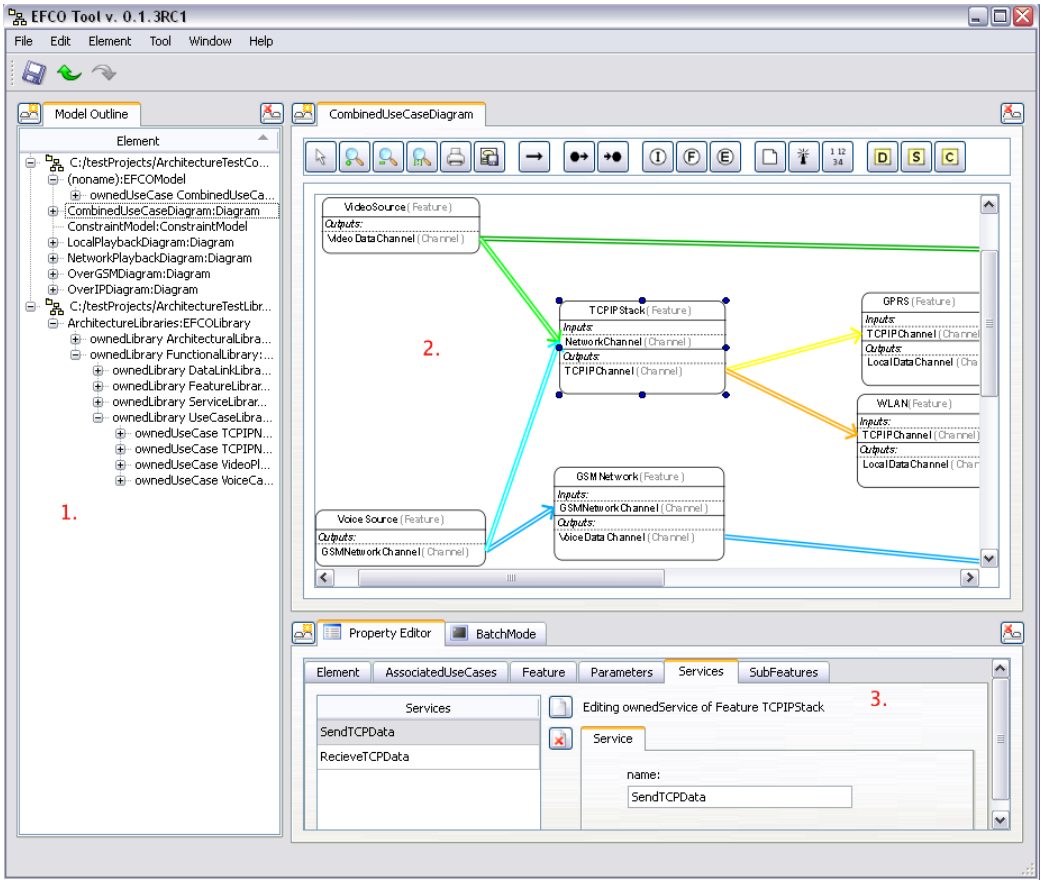

Figure 7: The EFCO Tool editing a Use Case diagram. 1. The outline editor. 2. The diagram editor. 3. The property editor

From the Use Case diagram it is at all times possible to zoom into a Feature to inspect how it has been composed from Services. This can be done by executing a context menu command while right-clicking on a Feature in the diagram editor. This generates a new Feature diagram that is displayed. Similarly as in the Use Case diagram, the compositon of existing Features can be changed by inserting new Services as copies from the library. Connections between these Services can then be added using toolbar commands or moved using similar gestures as in the Use Case diagram.

To edit more advanced information of the various elements in the models, the EFCO tool has a set of *property editors*. These property editors are organized in tabs below the diagram editor in the EFCO tool. Each element in the EFCO meta-

model has one or more associated property editor, which are shown when a model element is selected in e.g. the diagram editor or the model outline. These property editors can be configured to edit textual properties, such as names or parameter values, as well as relating or creating elements. While these editors are configured declaratively using a dedicated language for defining property editors, some elements, especially EFCO Projects and Use Cases have additional editors or wizards that directly concern the actual workflow, which have been programmed using the Coral programming interface. Examples of these are the editors for importing new CoFluent projects into the library.

When the Use Cases in the EFCO project has been designed or reviewed, a Simulation Use Case can be generated based on the Use Case. This functionality can ve invoked from the context menu. This function transforms the Use Case into a Simulation Use Case and inserts additional parameters and routing information necessary for simulation. The tool allows the developer to review the Simulation Use Case and make adjustments like e.g. changing parameters. The Simulation Use Case editor is otherwise similar to the Use Case editor.

When the developer has reviewed the Simulation Use Case, a new CoFluent Studio project is generated and exported to a file. This file can then be loaded into the CoFluent tool and simulated.

5.3 Constraint evaluation

The EFCO modeling language is restricted by the corresponding metamodel. The tool framework is constructed such that it is not possible to violate by constructing a model which does not conform to the metamodel. However, not all models that conform to the metamodel will correspond to a program that can be simulated in the toolset. Although some limitations or constraints can be considered straightforward with some understanding of the problem domain, some constraints can be imposed by the library components themselves. If a constraint is violated, it may result in a system that is not functional under all cicumstances.

To assist the developer during the design process, the EFCO editors use a *constraint avaluation engine* which is configured by a set of constraints. The constraint evaluation engine is based on an approach where models are continously checked for violations againts the constraints. The set of constraints is loaded automatically into the constraint evaluation engine as a model, can easily be extended by new constraint models. Constraints can be defined either by defining a graphical query using a declarative query language CQuery [10], or as Python scripts, supporting OCL-like [18] queries. The CQuery language supports a set of graph operators, including a negative matching operator and the *star region*, providing the possibility of using hierarchy and recursion in a declaratively defined query.

When the constraint evaluation engine detects that an error has been introduced by the user, a reporting facility will produce a report containing which

constraints are violated or held. In the case where a constraint is violated, the offending elements are reported. In addition to this, a suggestion for correcting the problem is presented. We have found that the impact on performance of using this component is relatively small. The benefits with this component, however, are clear, as the developer is notified of mistakes as the model is being designed.

6 Conclusion and Future Work

In this article we have presented a design flow which uses a modelbased approach to enable efficient re-use of design assets and automatic combination of features and their evaluation on different platforms. Further, the approach enables the designer to try different mechanisms for balancing the resource contention in the system by allowing features to have flow control constructs. Such properties are important for exploring what how a set of use cases need to be controlled in order to work properly.

The EFCO tool was implemented as an extension to the Coral Modeling Framework. The Coral Modeling Framework provides framework for defining new modeling languages, model transformation programs and many different types of editors, including diagram editors, form-based editors and interactive shell. Most of the editor components in Coral are defined declaratively by using tool-independent definitions. This has made it possible to reduce the time required to provide tool support for EFCO. Since updates on diagrams are decoupled from changes in the abstract model using DIML and the diagram reconciliation mechanism, it has been possible to integrate many different editor components editing the models while diagrams are up-to-date at all times.

As future work we will experiment with how different analysis methods can be used to complement the simulation. Such concepts as real-time calculus [5] could for example be used to calculate the performance bounds of the system. This kind of analysis is needed as simulations never show every possible scenario the system might experience. If the bounds of the system can be calculated it is possible to show that no task will miss a deadline. We also need to formalize the flow control concepts as this would enable the designer to calculate tight bounds of the controlled parts of the system and it would also enable appropriate control structures to be derived from the specifications of the system.

References

[1] Cofluent design homepage, available at http://www.cofluentdesign.com, 2008.

[2] Marcus Alanen. *A Metamodeling Framework for Software Engineering*. PhD thesis, Åbo Akademi University, May 2007.

[3] Marcus Alanen, Torbjörn Lundkvist, and Ivan Porres. Creating and Reconciling Diagrams After Executing Model Transformations. *Science of Computer Programming*, 68(3):128–151, Oct 2007.

[4] Jean-Paul Calvez. *Embedded Real-Time Systems. A Specification and Design Methodology*. John Wiley and Sons, 1993.

[5] Samarjit Chakraborty, Simon Kunzli, and Lothar Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10190, Washington, DC, USA, 2003. IEEE Computer Society.

[6] Johan Ersfolk, Johan Lilius, Jari Muurinen, Ari Salomäki, Niklas Fors, and Johnny Nylund. Design complexity management in embedded systems design. In *Proceedings of the 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB 2009)*. CEUR Workshop Proceedings (CEUR-WS.org), October 2009.

[7] Peter H. Feiler, David P. Gluch, and John J. Hudak. The architecture analysis and design language (aadl): An introduction. Technical report, CMU/SEI, 2006.

[8] Niklas Fors. Efficient combination of reusable components in embedded system design. Master's thesis, Åbo Akademi University, Faculty of Technology, 2008. http://research.it.abo.fi/research/ese/projects/efco/fors.pdf.

[9] Bart Kienhuis, Ed F. Deprettere, Pieter van der Wolf, and Kees A. Vissers. A methodology to design programmable embedded systems - the y-chart approach. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS*, pages 18–37, London, UK, 2002. Springer-Verlag.

[10] Johan Lindqvist, Torbjörn Lundkvist, and Ivan Porres. A Query Language With the Star Operator. In *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, volume 6(2007) of *Electronic Communications of the EASST*, Braga, Portugal, March 2007. EASST.

[11] Torbjörn Lundkvist and Ivan Porres. Coordination of Model Transformation Engines and Visual Editors. In Jari Peltonen, editor, *Proceedings of NW-MODE'09*, pages 269–283, August 2009.

[12] Johnny Nylund. Efcotool - a tool to efficiently combine and reuse components in embedded system design. Master's thesis, Åbo Akademi University, Faculty of Technology, 2008. http://research.it.abo.fi/research/ese/projects/efco/nylund.pdf.

[13] Object Management Group website. http://www.omg.org/.

[14] OMG. UML 2.0 Infrastructure Specification, September 2003. Document ptc/03-09-15, available at http://www.omg.org/.

[15] OMG. Unified Modeling Language: Diagram Interchange version 2.0, June 2005. OMG document ptc/05-06-04. Available at http://www.omg.org.

[16] Guido van Rossum. The Python Programming Language. Available at http://www.python.org/.

[17] W3C. Scalable Vector Graphics (SVG) 1.1 Specification, January 2003. Available at http://www.w3.org/TR/SVG11/.

[18] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

List of Figures

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3–5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 978-952-12-2429-4
ISSN 1239-1891