Design Complexity Management in Embedded System Design

Johan Ersfolk[1,2], Johan Lilius[2], Jari Muurinen[3], Ari Salomäki[3], Niklas Fors[2], and Johnny Nylund[2]

[1] Turku Centre for Computer Science, Turku, Finland
[2] Department of Information Technologies
Åbo Akademi University, Turku, Finland
FirstName.LastName@abo.fi
[3] Nokia Devices, Finland
FirstName.LastName@nokia.com

Abstract. Research on embedded system design typically focus 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 on all platforms. In this paper we address this problem by 1. proposing a structure for the design space, 2. an automated algorithm that generates performance models by combining use case models, and 3. an approach for performance optimization by adding flow control elements into the system design.

1 Introduction

Existing embedded system design methodologies focus on design space exploration in the architecture platform space. That is to say, they assume that the set of applications is fixed and a suitable architecture for this set of applications needs to be explored. In the mobile phone industry the design problem is often quite different and the situation is usually that there is a number of fixed platforms for which new applications are being developed using libraries of existing software components. This often leads to a situation where the concurrent execution of a set of applications needs to be simulated on a number of architecture platforms in order to analyse the resource sharing between the applications. In order to make the evaluation of such designs efficient there is a need for exploring how existing design methodologies and tools can be extended with functionality that addresses the problem of efficiently combining software components. In this paper we approach this issue with a model driven approach using our metamodeling tool Coral [1].

The design flow depicted in figure 1 highlights the communication between a system architect and the teams working on the different subsystems. The typical scenario in which this design flow is instantiated is when a set of new use cases needs to be implemented. This would involve for example adding video playback (a use case) and

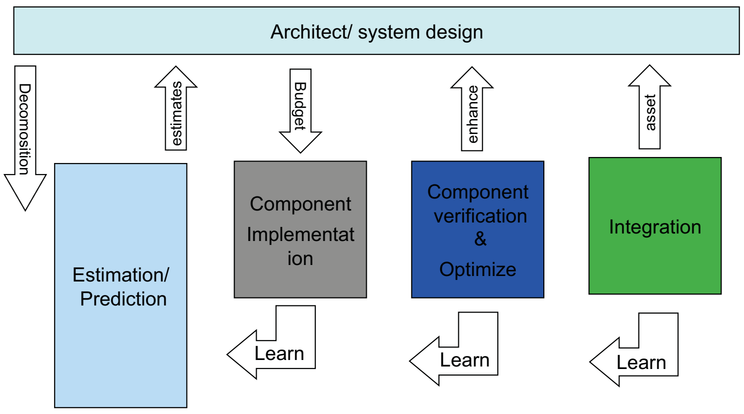

Fig. 1. A design flow and its feedback loops

video recording capabilities (a second use case) to a phone. The design will not proceed by trying to build a new system from scratch, but instead the goal is to find the minimal changes to an existing architecture to implement the new use cases.

The design would proceed approximately like this. The system architect takes the new use case and decomposes the system 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. At this point in the design flow it is important to obtain quick results. Therefore the individual elements in the system model are often quite abstract and focus only on the performance characteristics.

It is a key requirement to be able to evaluate different *use case combinations* quickly. Most often all use cases are not used at the same time, e.g. video playback might not be used at the same time with a voice call, if the phone does not support video calls, but video playback might happen at the same time as a file download. Therefore, it is important to know which use cases can be used in combination and to analyze the combinations for potential performance bottlenecks. The performance bottlenecks are typically caused by use cases sharing resources. In some cases it is not possible to run a specific use case combination on a platform which means that the platform needs to be modified. More often the problem of sharing a resource is due to stochastic behavior of data streams and badly tuned mechanisms for handling the resource contentions. In order to resolve resource conflicts and find the optimal parameters for the system the system architect can use different flow control mechanisms. In section 5 we describe different techniques and how these can be used.

When the system architect has found parameters that fullfill the performance requirements the design flow continues based on this validated information. 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 proven 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 and efficient (i.e. rapid) tools, based on dedicated analysis models, to support him in his design work. In section 4 we explain how such models can be obtained automatically. The requirement for the automatic use case combination is that the models have a specific structure, and such structuring mechanisms will be presented in section 3. In section 6 we shortly present a front-end tool (EFCO Tool) to CoFluent Studio [2].

Due to space limitations many pictures and details have to be omitted. A more thorough description of the approach can be found in [3, 4].

2 Related Work

There is a large number of tools and methods for design space exploration and it is not possible to mention all of them here. We will focus on the most relevant approaches and we will also describe the approach used in CoFluent Studio and compare it to our. In [5] several different methodologies intended to be used in the field of system-level design are discussed and compared.

The Y-chart approach [6] 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. The Y-chart approach does not specifically deal with software reuse in any way. If effectively separates architecture design from application design, but it does not guarantee that the applications can be reused easily for architectures or combined with other applications. What the Y-chart approach does for software reuse is that it specifies a structured method to map a set of applications to an architecture and simulate the results.

Another approach that focus even more on component reuse in the hardware part of the system is Platform-based design [7]. Platform-based design is an approach to embedded system design where refined specifications meet with abstractions of possible architecture implementations [7]. Platform-based design identifies well defined layers in the design process where the abstractions and refinements are done. Each abstraction layer must give enough information about lower levels of abstraction upwards, so that design space exploration can take place. Furthermore, constraints from higher levels of abstraction need to be passed down to lower levels of abstraction so that the refinement process can take place between layers. The difference to our aproach is that the set of applications is assumed to be fixed and that a suitable architecture for this set of applications needs to be explored.

CoFluent Studio [2, 8] is an embedded system design tool that enables performance analysis of hardware/software systems, by using the Y-chart approach. Consistent with the Y-chart approach, CoFluent Studio separates the functional model of the system from the architectural model of the system. By separately describing an application model and a platform model, and then mapping the two models together, an architecture model can be obtained. CoFluent Studio supports simulation of these models through automatically generating a SystemC test bench for performance analysis.

Functional design in CoFluent Studio is done by specifying a functional model of the complete system using a combination of a graphical notations and C code. The functional model can graphically be represented using structural and behavioral components called “functions”. Structural functions can contain other structural functions as well as behavioral functions. Behavioral functions specify a set of operations and their temporal ordering for a specific functional behavior. Communication between different functions is described using different communication components, which include communication channels, shared variables and events. The CoFluent functional model describes the systems behavior and timing without platform constraints and can be used to simulate the system without a platform. This can be used to analyze shared resources without being distracted by problems related to mapping. Our approach makes use of the methodology in CoFluent Studio but refines it by hiding details from the designer and by providing tools for use case combination and flow control.

The Architecture Analysis and Design Language (AADL) [9] 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 simulator generated by CoFluent Studio.

Real Time Calculus (RTC) [10] is an interesting approach to investigate schedulability and resource usage of real-time systems. RTC could be used instead of simulation to analyse our models. Multi-mode RTC could be used to analyse flow control. RTC is used to get similar information about the system as we are interested in.

In general, the difference between our approach and other related approaches is that our model describes and focus on the use cases of the system as the main modelling concept. Our model is also designed to allow easy combination and evaluation of use cases and provides an automated method to do this. Another difference is that we are not searching for an optimal platform for the system but investigating how to enable new use cases on an existing platform.

3 Asset management

In order to make the design phase efficient we need support for managing reusable assets and methods for combining these with new components that make up the new system. The goal with asset management is to provide a better way to support the life-cycle management of product families and the strategic decision making by enabling

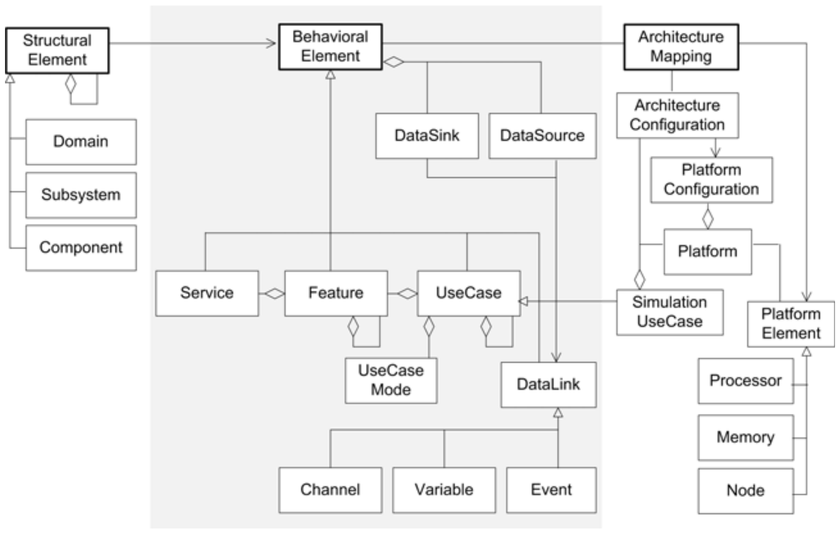

Fig. 2. The structural metamodel (left), the behavioral metamodel (middle) and the platform metamodel (right). The metamodel has been simplified in order to increase readability.

quick evaluation of new product features in order to give feedback for the business decisions.

We provide a set of structuring concepts encoded into a metamodel. The metamodel can be split into 3 parts. 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 paper we concentrate on describing the behavioral model as it is central to our modeling approach. More detailed information about the models can be found in [3, 4].

Although we use the term behavioral to characterize the part of the metamodel used to structure the functionality of the design, we do not propose a new approach to describe the functionality of atomic elements, 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).

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. For the video playback 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.

It could be argued that this distinction is ad-hoc and driven by what is possible to do in the CoFluent Studio tool, but there is a however a more abstract characterization of the concepts by looking at the communication characteristics of the concepts.

– A use case defines the structure and the dynamics of the communication with external actors.
– A use case mode defines a particular instance of a use case, and fixes the communication network topology. A use case mode can contain detailed descriptions of the internal structure of the network.
– A feature describes dynamic communication aspects between nodes.
– A service defines the node level routing.

We still need 3 other concepts into our meta-model:

DataLinks represent functional communication elements. The types of communication elemenents we use exactly corresponds to the elements in CoFluent Studio. A *channel* is a communication channel which must have a specific type. A *variable* is a shared variable between Features or Services, and it must also have a specific type. An *event* is a trigger for a communication element and has no type definition.

Parameters are values that have to be specified in the use case, but usually are given concrete meaning on the level of feature or service. Typical example parameters would be the frame-size of a picture, frame-rate. Parameters are not separately represented as an entity in the metamodel, but are instead given as attributes.

Actors are used to create external inputs for the simulation model. Currently the tool implements very simple actors, like a file-reader, and random number generators. In the future we plan to include network simulators for TCP/IP and other protocols. A more detailed description of the actor mechanism can be found in [4].

4 Use-case based evaluation

Under the assumption that the design of all systems is structured according to the meta-model presented in the previous section, the design of a new system can now proceed as follows. The new system will consist of an old system structured as a set of use cases and a set of new uses cases. Then the first questions 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. It is therefore important to be able to analyze use case combinations using different parameters rapidly. To this end we have developed a simple graph merging algorithm that given two use case modes (or features) creates a new model, that contains the behavior of both use case modes. The new model will contain shared elements and can

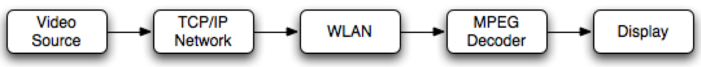

Fig. 3. Video playback use case

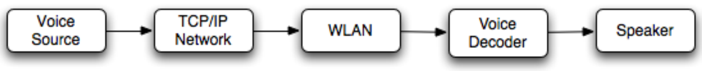

Fig. 4. Voice call use case

therefore be analyzed for problems in resource contention. The detailed description of the algorithm can be found in [3].

As a simple example we use two use cases: video playback and voice call. Figures 3 and 4 show the separate use cases using the notation in the EFCO tool and figure 5 shows the combined use case. The evaluation problem comes down to the sharing of the TCP/IP and WLAN features. This model can now be used to evaluate how the combined use cases can coexist on the given platforms.

As we combine *use cases* such that common functionality is merged there is a need to add mechanisms for routing messages through the system. This is done by adding a header to messages and adding routers to the design before the *simulation use cases* are generated and exported to CoFluent Studio. We need routers in two different situations, 1) for sending data through the choosen 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 routers analyse 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.

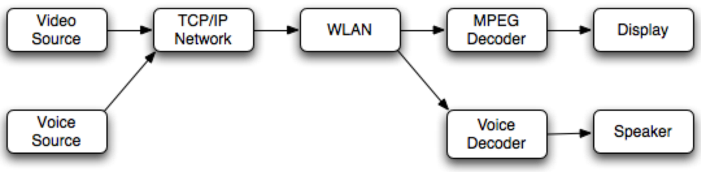

Fig. 5. Combined voice call and video playback use case

5 Flow control

A central observation that can be made is that in the presented approach the traditional distinction between a platform independent and a platform dependent model does not properly exist. Indeed the goal is that all the assets used in the models contain as much platform dependent information as possible. 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 advantage of this is that a single use case can be verified very much on its own. The only real issue that is left and that needs the platform mapping is the resolution of resource contention. This is the topic of this section of the paper.

When several use cases are running concurrently there is often a need to control the resource usage of some critical parts of the system. Problems with resource sharing are typically a result from a task using a resource, such as a processor or a communication channel, for long time intervals or of high buffer levels which lead to long message delays. Buffer delays can be a problem when the buffer is shared between use cases, it is then possible that a critical message gets stuck in the back of the message queue. Such problems could be solved by giving some messages higher priority but this would be a static solution and it would not take the state of the system into account.

In the example use case in the previous section components such as the video decoder might need control mechanisms that prevent its buffers from overflowing or underflowing. This kind of control is needed as most systems contain components with stochastic behavior and therefore adds burstiness or jitter to the system. Examples of such components are communication networks, DMAs, storage systems etc. In the example use case the TCP/IP and WLAN components will shape the data flow and it is essential for the simulation results that the behavior of such components can be modeled and that the impact of these is considered in the simulation.

Furthermore, two use cases might also share a resource such as a processor due to the mapping. In this case flow control can be used to restrict the processor usage of one or both of the tasks, the desition of which tasks are allowed to use the processor can be made based on such properties as buffer level. As an example consider a task with bursty input, the task will alternate between periods of high activity and periods of being idle, this will in turn affect the execution of the other tasks running concurrently on the processor. The other tasks will experience periods when these get more or less processor time. Depending on the length of the periods and on the execution times of the tasks, some tasks might miss deadlines during the periods when the first task is active. This problem can be solved by restricting the processor usage of the first task, e.g. by suspending the task when its output buffer has reached a certain level, as a result the execution of the task will not follow the burstiness of the input stream anymore but instead the periods can be made shorter and the processor usage more even. In order to find the optimal parameters for the system the designer can try different flow control mechanisms.

In our approach the designer can add flow control constructs to the system in order to balance the resource contention. In its simplest form flow control is a sender and a receiver feature where the receiver monitors a buffer and sends feedback messages to the sender when the buffer has reached one of the defined levels. The sender then

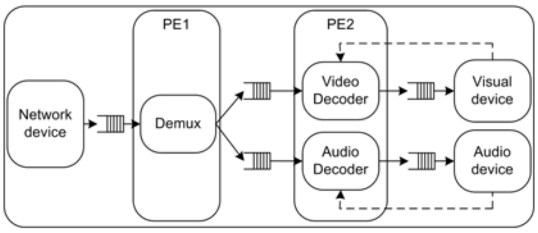

Fig. 6. An example of flow control

suspends or resumes its services depending on the content of the feedback message. Except for tuning the buffer levels, such constructs affect the resource contention as it can give the resource to a task that really needs it at the moment. In general the desision to add flow control to the design is based on the designers experience, therefore, the proposed approach does not consider how to get the optimal system but describes a methodology how to design a system and what kind of tools the designer needs. By using different types of flow control the designer can solve problems related to shared resources which in some cases means that he can avoid to make modifications to the platform.

In figure 6 a system running an audio and a video decoder is presented, the system has a general purpose processor (PE1) which in this case handles the network interface and feeds the streams to the decoders, the general purpose processor also runs the operating system. The actual decoding is performed on a digital signal processor (PE2). As the input streams are received over a network and because the sender serves several client simultaneously, the input streams are bursty. The resource sharing we need to simulate is 1) the sharing of the network and 2) sharing of the processor used for decoding the audio and video streams.

The goal of the system architect is to find the parameters that ensure that the playout buffers of the audio and video decoders does not underflow. If either decoder process is late, i.e. the level of the playout buffer is low, it should get more processor time than the other tasks. For this purpose the buffers are controlled using a flow control mechanism which changes execution rate of the tasks depending on the level of the playout buffer. In figure 6 the feedback channels are illustrated with dashed arrows. What can be ensured with this type of simulation is that if we add a specific flow control, the system will work as long as the input streams are within the limits we have specified regarding burstiness and jitter.

In this example the flow control is simple in that sense that it directly controlls the output buffer of the sender feature. Often this is not the case, in many applications control messages travel in the same channels as other messages and the messages might pass through several features, as an example consider a sender and receiver communicating through a network. This will affect such aspects as the delay of feedback messages and the delay before the impact can be measured after changing the state of a feature. If there are several buffers between a sender and a receiver, the level of the receivers input buffer might continue to rise for some time after the sender has been suspended. Such properties can be analyzed when simulating the system with different parameters and by monitoring buffer levels and message delays.

During the simulation stage buffer levels and message delays at different parameter settings can be measured for the use case combinations. The designer obtains useful information about the control mechanisms needed for specific use case combinations. It is essential that flow control parameters can be easily modified and that the structure of the flow control mechanisms are general enough that the design does not need to be changed when changing type of flow control. This enables the designer to try several setups rapidly and find a solution that satisfies the requirements.

In our metamodel flow control controlls the features by suspending and resuming the services in these. In the generated CoFluent project a controller is added to the output of a feature and the services of the feature is connected to the controller by suspend/resume control signals. A controller is a service that forwards data in zero time if the output channel is not full and it has no buffer space on its input. This is important as adding a controller should not add buffer space or delay to the system. Further, the controller has an input port for feedback messages and an output port for setting the state of the other services of the feature. The feedback messages are produced by an observer located in the receiver feature. The observer records buffer levels and number of messages received and sends feedback messages according to the flow control protocol choosen. The observer has similar features as the controller as it does not add delay of buffer space to the system. Feedback messages are handled as any other message and can in some cases travel on the same channels as the data, it is only at the sender or receiver *features* where the messages need to be routed to the corrects data source/sink. This simple structure of flow control allows different feedback based flow control types without changing the structure of the model.

The basic flow control types we have implemented in the EFCO-tool are Watermark based flow control, Xon/Xoff, Window based flow control and Credit based flow control. These basic flow control mechanisms are based on existing protocols used in computer communication.

Watermark based flow control The Watermark based flow control makes its decisions based on available buffer space at the receiver. The sender can decrease/increase its transmission rate depending on if the buffer has reached its low or high water marks. The transmit rate can be changed for example by changing the priority of the sending process.

The **Xon/Xoff** protocol is similar to watermark based flow control but is simpler as it either signals the sender to stop or continue sending jobs based on the input buffer size of the receiver. The controller in the sender feature receives the feedback message and suspends/resumes its services depending on the content of the message.

A different type of flow control is **Window based flow control**. In this type of flow control the senders is allowed to transmit a given number (window size) of messages before acknowledgments are needed. This means that it is the number of messages between the sender and the receiver that is controlled and not the number of elements in a specific buffer. Window flow control is widely used; one example of a refinement of it is the sliding window protocol used in TCP.

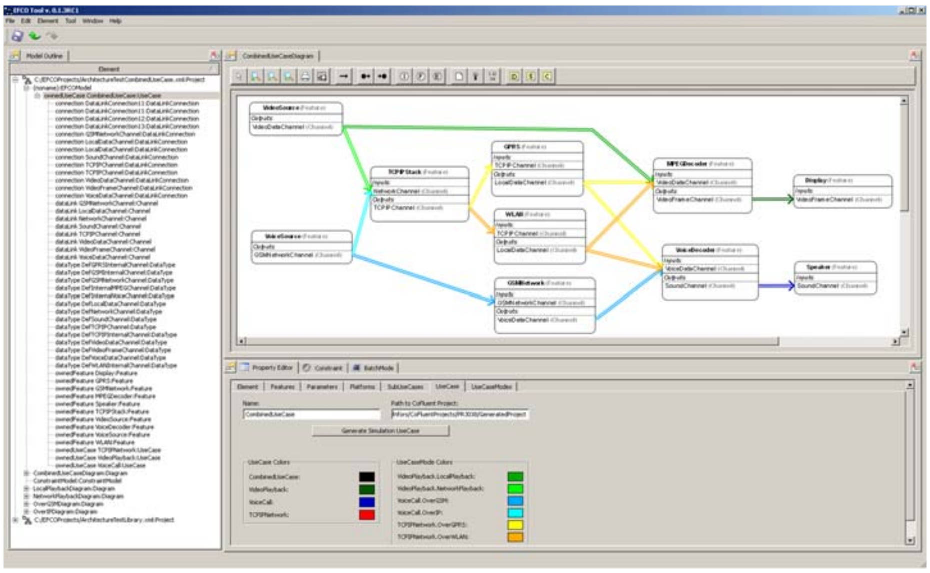

Fig. 7. Overview of the EFCO tool

A similar protocol is the **Credit based flow control**. Here the receiver gives credit to the sender which indicates the number of messages the sender is allowed to send, the sender consumes this credit while sending messages.

The flow control protocols described here are abstract simulation tools that gives the designer guidelines to how parts of the system needs to be controlled in order to make the whole system stable. The approach does not specify how the flow control should be implemented in a real system, instead this is left to the designer. In real systems flow control could be implemented within a single application be having the application suspend/resume itself depending on some criteria, in this case adding flow control would only make local changes to the application and not to the system. Another solution is to implement flow control support in the operating system, it would then be possible to have the scheduler make its decitions based the state of the use cases.

6 EFCO Tool

The EFCO tool is built on the Coral framework [1]. Figure 7 gives an overview of the tool, and shows the three most important parts of the tool. These are all part of the Coral modeler, and only support for the EFCO modeling language was needed to implement in them. The outline editor (to the left) shows the current loaded models in a tree-like hierarchy, both libraries and projects are loaded and shown in this editor. Elements in this tree-view can be selected, copied and pasted, and also dragged and dropped into other parts of the tool. The diagram editor (right top) shows a diagrammatic representation of different parts of the models loaded in the outline editor. Elements from one model in the outline editor can be dragged and dropped into the diagram editor of another model.

For example, a library element can be dragged from a library in the outline editor into a diagram of a project, which effectively combines the instantiated library element into the project. The property editor (right bottom) presents different editable properties of the currently selected element.

The EFCO tool implements the use case based metamodel and provides a set of tools for manipulating it. It supports import from and export to CoFluent Studio projects, this is important as services are the lowest level of detail handled in the EFCO tool and the implementation of these must be done in CoFluent Studio. The EFCO tool does not include any simulation tools and therefore the projects must be exported to CoFluent Studio, this because EFCO tool has only been implemented as a tool for reusing and combining components in an efficient way. What EFCO tool provides to the design is tools for managing use cases. It provides the nodes as a tree which enables easy reuse on any level of the design. It is possible to reuse whole use cases but also features and services. The tool also implements use case combination tools, the combining of components should be done in such a way that common smaller components are recognized and not duplicated, and mappings to architecture elements should also be reused if they have already been created. If a design already contain platform mapping the tool also keeps the mapping as it is usually only new use cases that should be mapped and the existing use cases will not be modified if not neccessary.

Before exporting the project to a CoFluent Project the use case is translated in to a *SimulationUseCase*. A *SimulationUseCase* contains everything needed to simulate different modes and combinations of the system, compared to a *UseCase* it breaks the structure suitable for reuse and implements components needed for simulating the system. To be able to simulate a *UseCase* created in EFCOTool it needs to be exported back to CoFluent Studio, before the exporting of the new *UseCase* can be done, it needs to be transformed into a *SimulationUseCase*. When transforming a *UseCase* into a *SimulationUseCase*, all routers needed for simulating the *UseCase* are automatically added to the model. Additional generic parameters are also added, one for every *SubUseCase* and one for choosing between different *UseCaseModes* in a *SubUseCase* or *UseCase*. Information regarding the architecture part of the model is also gathered to make the exporting mechanism easier.

There are currently two different types of routers, namely *UseCaseRouters* and *UseCaseModeRouters*. An *UseCaseRouter* is needed if, for example, a project element has more than one connection from the same output to project elements in different *SubUseCases*. The generic parameter that is automatically created for each *SubUseCase*, can be used to enable or disable *SubUseCases* when simulating the model. When disabling a *SubUseCase*, all data to that *SubUseCase* is routed to a discard channel in the *UseCaseRouter*. An *UseCaseModeRouter*, on the other hand, is needed if there are more than one connection from the same output in a project element to other project elements in the same *UseCase* or *SubUseCase*. Choosing the active *UseCaseMode* in a simulation is done via a generic parameter, which is automatically created when the *UseCaseModeRouter* is created.

The EFCOTool also supports the ability to automatically run several executable CoFluent Studio simulation models, with certain specific parameter values. This is called a *BatchMode* run, and it is located in the *BatchMode* tab in the *property edi-*

tor area of the EFCOTool. To be able to run a *BatchMode* in EFCOTool, a XML-file (that follows a XML-schema) needs to be created. This XML-file contains all information describing the different simulations, along with their parameter and simulation configurations. The batchmode tool is important for the design space exploration as the designer can analyse a large number of parameters and use case modes and combination without interacting with the tools. The designer can then after the batchmode run check the simulation results for feasible solutions.

7 Conclusions and Future Work

In this paper we have introduced an approach that uses a hierarchy of concepts that help structure the components/assets that are needed for composing the new product and map these to requirements on the business level. This approach uses the use case as the fundamental high-level design concept and structures the design so that it is easy to store design assets into libraries. This also enables use of the automated algorithm for creating new performance analysis models based on a set of use cases which allows for automatic combination of features and their evaluation of different platforms. Further, an approach for finding parameters to control the resource contention in the system by allowing features to have flow control constructs was also introduced. Such properties are important for exploring what kind of control a set of use cases need in order to work properly. Except from optimizing performance such construct can solve resource sharing problems and it might be possible to avoid to modify a platform.

Currently the approach is tied to the CoFluent Studio tool, but in principle any SystemC evaluation framework can be used, since the algorithms are all implemented on the level of the meta-model. As future work we will experiment with how real-time calculus [10] could be used to calculate bounds of the system. This is useful in case of realtime systems as the simulation can never show every possible special case; if the bound can be calculated it is possible to show that no task will miss a dealine. Simulation will still be useful for studying the performance of the system. Flow control can also be studied using RTC, one suitable method that can be used is multimode RTC [11]. It is also possible to directly describe the flow control mechanisms using RTC as long as we are only interested in the best case or the worst case, examples of this can be found in literature concerning network calculus [12] which has been used to analyze networks that are based on window buffer protocols.

References

1. Lundkvist, T., Porres, I.: Coordination of Model Transformation Engines and Visual Editors. In Peltonen, J., ed.: Proceedings of NW-MODE'09. (2009) 269–283
2. Cofluent design homepage, available at http://www.cofluentdesign.com (2009)
3. Fors, N.: 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.
4. Nylund, J.: 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.

5. Živković, V.D., Lieverse, P.: An overview of methodologies and tools in the field of system-level design. In: Embedded processor design challenges: systems, architectures, modeling, and simulation-SAMOS, New York, NY, USA, Springer-Verlag New York, Inc. (2002) 74–88
6. Kienhuis, B., Deprettere, E.F., van der Wolf, P., Vissers, K.A.: A methodology to design programmable embedded systems - the y-chart approach. In: Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS, London, UK, Springer-Verlag (2002) 18–37
7. Sangiovanni-Vincentelli, A.: Defining platform-based design. EE Design (2002)
8. Calvez, J.P.: Embedded Real-Time Systems. A Specification and Design Methodology. John Wiley and Sons (1993)
9. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis and design language (aadl): An introduction. Technical report, CMU/SEI (2006)
10. Chakraborty, S., Kunzli, S., Thiele, L.: 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, Washington, DC, USA, IEEE Computer Society (2003) 10190
11. Phan, L.T.X., Chakraborty, S., Thiagarajan, P.S.: A multi-mode real-time calculus. In: RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium, Washington, DC, USA, IEEE Computer Society (2008) 59–69
12. Le Boudec, J.Y., Thiran, P.: Network calculus: a theory of deterministic queuing systems for the internet. Springer-Verlag New York, Inc., New York, NY, USA (2001)