Assembling Components from Requirement Specifications

Amrit Panda, Onur Derin
ALaRI, Faculty of Informatics,
Universita della Svizzera Italiana (USI)
{*amrit.panda@lu.unisi.ch, onur.derin@alari.ch*}

1. Introduction

Component based design has proved to be an effective approach in the development of large and complex software systems. Extending this methodology to the design of embedded systems requires identifying and assembling reusable components that capture the behavioral properties of the system which are independent from the implementation. One of the most important benefits here is to provide the designer with very high level semantics of the components and let the framework care about creating communicating channels between components and satisfying extrta-functional properties. In this design process from the specifications to the final synthesizable blueprint ew focus on the composition of components given a set of specifications and a repository of components. In order to validate the correctness of our composition algorithm we emphasize on the use of executable specifications at different levels of abstraction viz. service, functional and non-functional levels. We also propose a very abstract way of specifying the system that would be the input for the composition algorithm and lead to the final design.

2. Related Work

Significant work has been done towards understanding the interplay between partial and complete specifications for composition of component by the group of Broy, Kruger and Meisinger [1]. The problem being addressed is the complexity of completely describing all behaviors of all the components involved in large and distributed systems. Instead a partial view of requirements on overall system is proposed. Services are used for partial specification of system and components for the complete specification. Abstracting a component into services that factor out collaboration among the software components required for fulfilling a certain task requires a better understanding of the interaction among the components and services. A component may offer a number of services and for a particular service; we are interested only in certain behavior of certain components. The above being said it is clear that a component behaves differently when considered within the scope of different requirements. Adhering to classical component design, the components are identified by their interfaces and interfaces of different components connected by channels. Formal definitions of components interfaces and services have been presented for better specifying their behaviors. The starting point is a stream which constitutes a channel history and interfaces are defined over channel histories. An interface can have a syntactic as well as semantic notion. A component inherits the definition of its interface. The syntactic definition concerns only the types of data stream that the channel allows whereas the semantic definition speaks more about the functional behavior of the component with the interfaces as a mapping of input and output sequences of streams. The systactic and semantic definitions enable composition of components from a Message Sequence Chart. Component reuse has been enabled by adding to the incremental and independent implementability theories of interfaces [4]. The research group of Prof. Gajksi at UCI is also examining the missing semantics on the design process and considering executable specifications. Their main focus is however on the accuracy and not on the composition of the components for the system. These models provide novel ways of describing components [2] [3]

3. Our Approach

The composition methodology proposed by the group of Broy roots from message sequence charts describing the behavior of systems and interactions among constituent components. However, a major concern in this process is the correctness and efficiency of this composition and catering to the non-functional requirements especially when the design targets real time and distributed embedded systems. Moreover message sequence charts are by themselves not the highest level of abstraction and their efficiency relies on the efficiency of requirement to specification transformations. We also consider appropriate model of computation for the composed system that would guarantee reliability and performance of real time systems. The entire design process from specification to the blueprint is shown in the figure below.

3.1. Methodology

We are developing an algorithm that uses the semantic interfaces for composition of services. However we do not

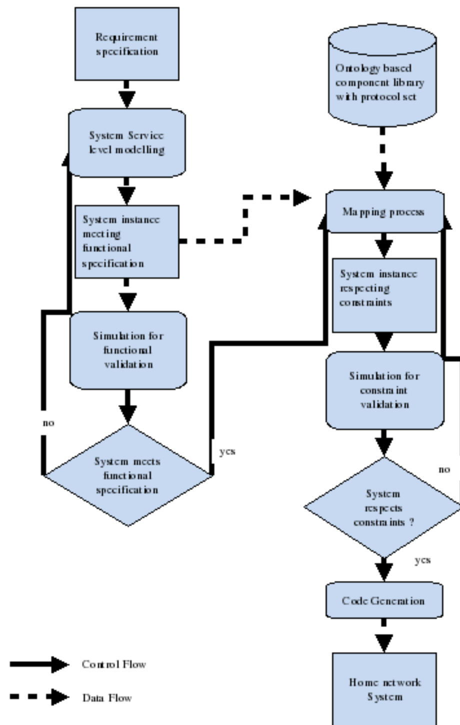

Figure 1. Design process

start from a Message Sequence Chart, rather we use a higher level of abstraction which we call the signature of the system. The signature is basically the requirements for the system rephrased formally in a restrictive language with additional qualifiers which can specify the time, power and cost constraints of the system. In a way the signature is the design entity closest to the requirements. The use of signature has a three-fold advantage over message sequence chart. Firstly, with a proper choice of specification language we can express real-time and distributed embedded systems where the non-functional requirements (time, power and cost constraints) are equally important as the functional requirements. One can argue that the non-functional requirements can be specified even in MSCs using annotations however one tend to forget that the annotations made in natural language results in ambiguities which are avoided by the restrictiveness of the signature. Secondly, the algorithm that performs the composition can directly make use of signatures without further modifications because of its formal representation. Above all, it provides a much higher level of abstraction thus freeing the designer from caring even about the preliminary design considerations. Once a signature of the system is identified we feed it to our algorithm that finds a set of efficient service compositions. However care should be taken here because the algorithm heavily relies on a repository which contains the available services and components with their formal definitions. This repository is indeed created by us and every service/component is associated with a pair of definitions capturing their behavior (syntactic and semantic). Given any service we can easily map this service to component interfaces and thus we have the system instance composed of components meeting the functional requirements. It is noteworthy that even if we compose a system pertaining only to its functional requirement the non-functional behaviors extracted from the signature are still retained for use as we move down the abstraction level. The main aim of this system instance is to be able to verify the correctness of the composition algorithm which implies that this model should be an executable model that could be simulated. It is this phase where we also take into account the model of computation to be used.

3.2. Simulation Environment

We have chosen the Ptolemy II framework by UCB for simulating our component model. This framework enables use of heterogeneous mixtures of models of computations that govern the interaction between components. It also allows graphical modeling of systems using the tool vergil and then simulating the system. The graphical model of any system in vergil corresponds to a XML ot MOML file. Our idea is to generate this .moml file as a result of the composition algorithm (after deriving the component network from service composition) and thus we would have an executable for verification.

3.3. Choice of model of computation

Ptolemy models are controlled by a director which implements the model of computation for the system. We are using the Component Interaction (CI) director, an experimental director, to simulate our functional model. The CI director uses a Push/Pull way of interaction. Each component is an actor in the language of Ptolemy and can be active or passive. Any kind of interaction within a model with CI director resolves around only two actions namely push and pull. However, it is not an actor that is characterized by push and pull, but its the interfaces which are either of kind pull or push. An actor with input pull interfaces or output push interfaces is an active actor in general. Each active actor is a separate thread in the simulation and all the passive actors are managed by a single main thread called the manager. Throughput being one of the dominating performance indices of RTES, we aim to leverage this index by efficient resource utilization of participating components. We allow only asynchronous communication among components thereby releasing their resources from awaiting other components with large response times. We are developing custom components as CI actors for the simulation which will fill the component repository. It should be clear that

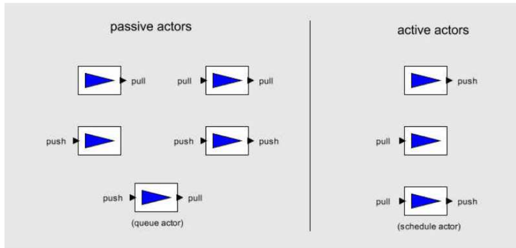

Figure 2. Active and Passive Actors

there are two domains within the repository. The first being only the pair of formal definitions of each component/service and the second being Ptolemy actors. It is the first domain that is used by the composition algorithm and the actors used solely for simulation using vergil. Ofcourse the components are the same for both domains. Shown in the figure is an example of a Ptolemy model using our custom actors.

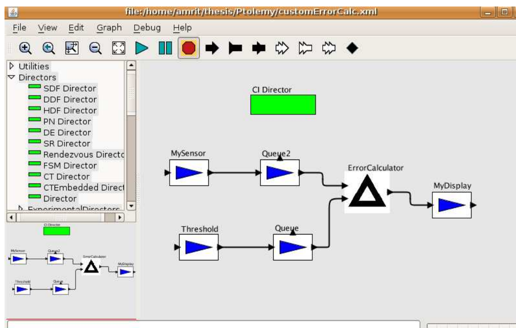

Figure 3. Simulation Model

4. Case Study

As a proof of concept we plan to deplot our methodology for smart home environments. We examine a simple control application for maintaining the temperature of a room. Since we are still in the process of defining a restrictive specification language and refining the algorithm, in this section we just present the comntrol algorithm and custom component model design for the simulation. The specification for this scenario in natural language is as follows: maintain the temperature of the room at 25 C, a tolerance of 1.8 C is permissible. We expect the composition algorithm to generate a .moml file involving the following components: a temperature sensor, an error calculator, a controller, a cooler (actuator) and a heator (actuator). Since we provide components with different characteristics in the repository, e.g there would be coolers that can have convective transfer coefficient up to a certain limit, we would ensure that the algorithm chooses all the available components for functional model and then later use the non-functional specifications (tolerance in this case) to prune the set of component network. On a service level these component interfaces would provide services which on an abstract level could be invoked remotely from another component. An important distinction needs to be highlighted here concerning the implementation of these components for real scenario and simulation purpose. Let us consider a controller invoking a sensor service. In this case the controller is definitely an active actor and the sensor a passive actor. So, in the simulation the controller will have an independent thread but not the sensor. Please note that in real scenario every component is independent of other components and there is no central controller, meaning every component irrespective of being active or passive has its own thread. The sensor for example in this case will have an independent thread that listens for requests seeking the sensor reading. The control algorithm that maintains the temperature of the room takes into account various factors that can lead to change in the room temperature. One major factor that effects the temperature change significantly is the convective transfer by walls/doors/windows which is a function of the difference of environment temperature and room temperature, the coefficient of walls and the area of the walls. We model the room as an entity (component) which has a variable convective transfer coefficient due to the fact that the opening of a door/window changes this coefficient. Our controller has two functions. The first function is to determine the coefficient of heat transfer for the actuator (heater/cooler) based on the room and external temperature and the room coefficient. The second functionality which is actually being modeled as a separate component is to estimate the present temperature assuming the room coefficient constant over the period form the last reading to present and then compare it with actual sensor reading. If the readings differ then it should determine the new room coefficient of the room and use it for the first functionality.

5. Status

This project is ongoing and right now we are considering various alternatives for a restrictive specification language and developing the composition algorithm at the same time. Concurrently we are also building custom components that could be used for simulation repository of components. For the non-functional requirements validation related to timing we propose to use a network simulator.

References

[1] Manfred Broy,Ingolf H. Kruger and Michael Meisinger, *A Formal Model of Services* ACM Transactions Software Engineering and Methodology: February, 2007.

[2] Nong Fan,Viraphol Chaiyakul and Daniel D. Gajski, *Usage-Based Characterization of Complex Functional Blocks for Reuse in Behavioral Synthesis* IEEE: 2000.

[3] Gwo-Dong Chen and Daniel D. Gajski, *An Intelligent Component for Behavioral Synthesis* 27th ACM/IEEE Design Automation Conference

[4] Laurent Doyen, Thomas A. Henzinger and Barbara Jobstmann, *Interface Theories with Component Reuse* EMSOFT: October, 2008