A Modular Method for Global System Behaviour Specification

Urooj Fatima and Rolv Bræk
Department of Telematics, Norwegian University of Science and Technology (NTNU), NO-7491, Trondheim, Norway

Keywords: Model-driven Engineering, Service Composition, Interface Behaviour, Activity Diagrams.

Abstract: The challenge addressed in this paper is how can we specify the global behaviour of distributed reactive systems in a way which eases the comprehension of the system without compromising its specification's correctness, completeness, modularity and readability. Instead of defining the global behaviour models in a monolithic way, we approach the problem by decomposing the specification into interface functionality and core functionality. The resulting interface-modular method for system specification is presented and discussed in this paper using a *TaxiCentral* as case study. The novelty of this method lies in the clear separation of interfaces from core functionality in global specification, and the use of activity diagrams in combination with collaborations to express and compose the specifications.

1 INTRODUCTION

Normally in system development, after requirements have been captured and analysed, the desired system behaviour is first specified from a global, cross-cutting, perspective involving several entities and then mapped to a design structure of components with precisely defined local component behaviours. The global behaviour emerging from the joint local component behaviours shall of course correspond to the specified global behaviour. Such correspondence can be assured in two ways: (1) by a process of verification, that is verifying after a design has been developed, that the local behaviours of the designed components are in accordance with the specification; or (2) by a process of design synthesis (transformation) whereby the local designs are derived from specifications in such a manner that correctness is guaranteed. Most current system design methods follow the first approach because fully automated design synthesis has not been practically feasible. There are two main reasons for this:

a. it is normally very difficult and/or impractical to completely specify global behaviours.
b. the global behaviour emerging from a direct mapping to a distributed design may contain undesired behaviours that follow from the nature of a distributed design and not from the specification itself. This kind of behaviour, sometimes referred to as implied scenarios, is the cause of so-called realizability problems.

In order to solve problem 'b' mentioned above it is necessary that all realizability problems can be identified and resolved as part of the design synthesis process. In (Castejón et al., 2007) the authors provide a classification of the various causes of such problems and explain how problems can be detected on the level of global behaviour and subsequently resolved during design. Based on this foundation (Kathayat and Bræk, 2011) has proposed some refinements to the analysis while (Kathayat et al., 2011) and (Fatima and Bræk, 2013) defined a method for design synthesis using activity diagrams for both global (source) and local (target) behaviour where the resulting local behaviour is in a form of activity diagrams that can be subject to extensive analysis before generating product quality (Java) code using a tool called Reactive Blocks (ReactiveBlocks, 2014). Thus, there is now a systematic way to overcome problem 'b' and principal solutions to enable highly automated design synthesis.

In this paper, we focus on problem 'a' mentioned above - how to achieve the necessary completeness, rigour and modularity in global behaviour specifications to enable highly automatic design synthesis in practice?

Sequence diagrams of some sort are probably the most used notations for global behaviour. They define behaviour in terms of interactions taking place among different components of a system and/or its environment, either in the form of asynchronous message passing or in the form of synchronous invocations. They are well suited for specifications since they consider systems and components from the outside and describe only their externally visible behaviour. Purely sequential behaviours are easy to specify completely using sequence diagrams. It is concurrency that causes problems. The service provided by a distributed reactive system normally involves several concurrent parts. The number of possible and relevant interaction orderings is then often beyond what is practically feasible to specify, and therefore completeness is not achieved. The general countermeasure for this kind of problem is to factor out concurrency and thereby reduce the number of interaction orderings needed to be explicitly modelled. Rather than considering global behaviours involving many concurrent parts, one may consider only two parts at a time and their interface behaviour, possibly decomposed further into smaller and more manageable sub-behaviours. This kind of decomposition follows naturally from the use of collaborations to structure and decompose global behaviours as we shall see in the following. The remaining problem then is to model the dependencies and global ordering among the interface behaviours.

Even when completely defined, interactions are not sufficient to fully specify the behaviour of systems and components. In many cases some internal data and data manipulation is equally important to the environment and the users of a system and therefore need to be specified as well. We are not considering internal design details here, only the data and operations that are important and give value to end users and other systems in the environment are considered.

From this we deduce that a complete specification needs to cover (at least) two related aspects: the external interactions and their ordering; and the internal data and operations that give value to the environment, which is called the core functionality in the following. There are (at least) two ways to organise such specifications:

1. the integrated approach in which the core functionality is embedded with interactions in one (large) specification,
2. the interface-modular approach where core functionality and interfaces are specified separately and then combined.

The second approach is the topic and main contribution of this paper (The authors have previously used the integrated approach and found it feasible, but hard and error prone to develop). The interface-modular approach is promising because it simplifies the process of developing complete specifications and also results in more modular specifications. As it turns out, the core functionality models serve both to specify the core functionality itself independently of particular interfaces, and as a glue taking care of the dependency among interfaces.

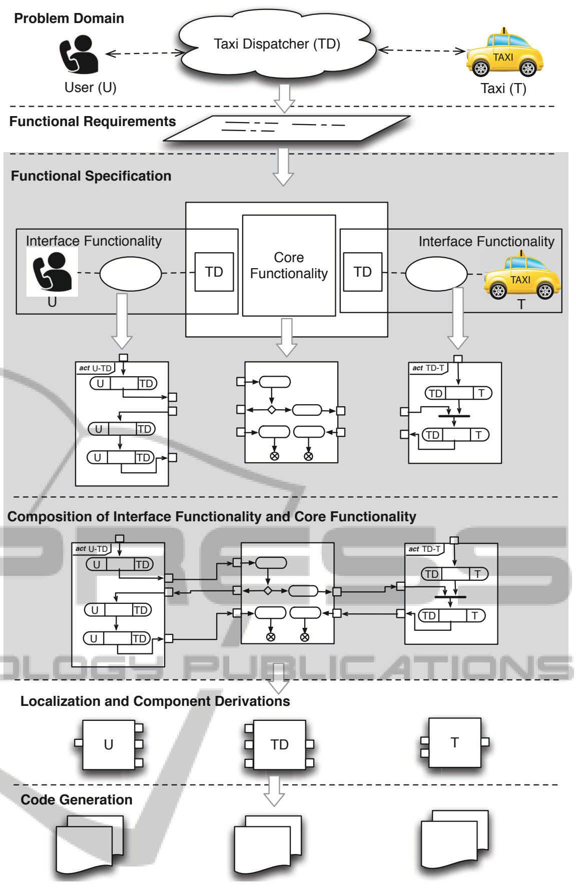


Figure 1: The overall illustration of the proposed method. The area highlighted with grey background is the focus of this paper.

The main properties of the method are the following:

- It simplifies achieving completeness in functional specifications.
- It covers interface functionality as well as core functionality.
- It defines interface functionality and core functionality in separate modules that are partly independent and may be composed in different ways.
- It specifies interface behaviours in a form that later can be used as contracts for lookup and validation purposes.

We use UML activities to define behaviour because they provide the ordering constructs needed for our purpose and allow us to stay within one notation for both global and local behaviours. Moreover, they support building blocks of variable granularity with well defined interfaces suitable for composing the interfaces and core functionality.

Figure 1 serves to illustrate the approach in terms of a *TaxiCentral* that shall be used as a running example throughout the rest of the paper. The rest of the paper is organized as follows. The functional requirements of the *TaxiCentral* are presented in Section 2. In Section 3, we present our interface-modular method by explaining the guidelines and applying them on the *TaxiCentral*. Section 4 explains our method validation. We discuss related work in Section 5 and conclude in Section 6.

2 FUNCTIONAL REQUIREMENTS

Functional requirements normally contain an informal textual specification of the core functionality (in terms of events, data and operations).

In the *TaxiCentral*, *Users* can book *Taxis* by placing taxi-booking requests to a *TaxiDispatcher*. *Taxis* inform the *TaxiDispatcher* about their availability. The *TaxiDispatcher* keeps an overview of available *Taxis* and assigns *Taxis* to *Users* as fairly as possible. Once a *Taxi* is assigned to a *User*, it can contact the *User* via phone call.

The *TaxiDispatcher* is driven by events generated by users and taxis and communicated across the interfaces as messages represented by the tokens in the activity diagrams. The functional requirements given below are organized according to the external events and specify the actions to be taken in response. Italicized items represent important domain entities and data. Italicized bold items represent important domain events.

A. ***TaxiRequest.*** The *User* initiates the taxi-booking request by sending a request to the *TaxiDispatcher*. As a result, either of the following two responses shall be generated:
   A1. ***TaxiAssign.*** If a *Taxi* is available, the *Taxi* shall be taken out of the *taxi queue* by the *TaxiDispatcher* and assigned to the *User*.
   A2. ***UserWait.*** If a *Taxi* is not available, the *User* shall be entered in a *user queue* and receive wait notification from the *TaxiDispatcher*, indicating the position in queue.

B. ***TaxiAvailable.*** The *Taxi* updates its availability status to the *TaxiDispatcher*. As a result, either of the following two responses shall be generated:
   B1. ***UserAssign.*** If a *User* is waiting in the *user queue*, the *User* shall be taken out of the *user queue* and assigned to the available *Taxi*.

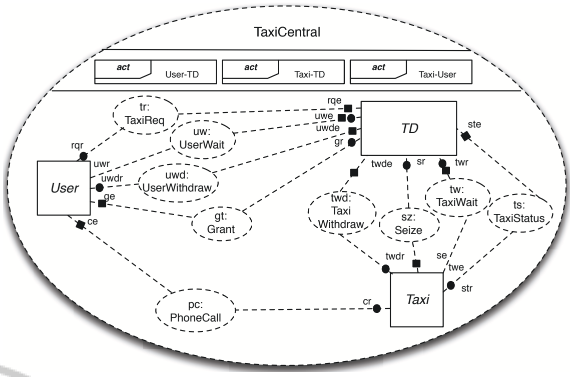


Figure 2: UML collaboration diagram showing roles and collaboration uses in interfaces of *TaxiCentral* service.

   B2. ***TaxiWait.*** If no *User* is waiting in the *user queue*, the *TaxiDispatcher* shall insert the *Taxi* reference in a *taxi queue* and send wait notification to the *Taxi*. The *Taxi* may receive its queue position.

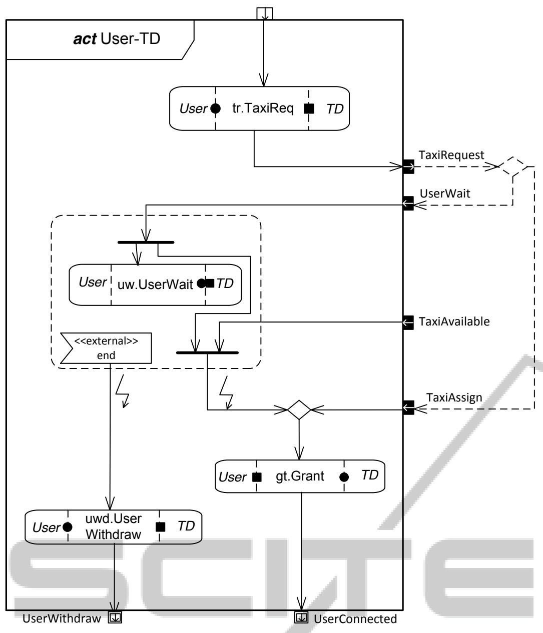
C. ***UserWithdraw.*** The *User* can withdraw its request while waiting in the *user queue*. Its reference shall then be removed from the queue by the *TaxiDispatcher*.

D. ***TaxiWithdraw.*** The *Taxi* can withdraw while waiting in the *taxi queue*. The *TaxiDispatcher* shall then remove its reference from the *taxi queue*.

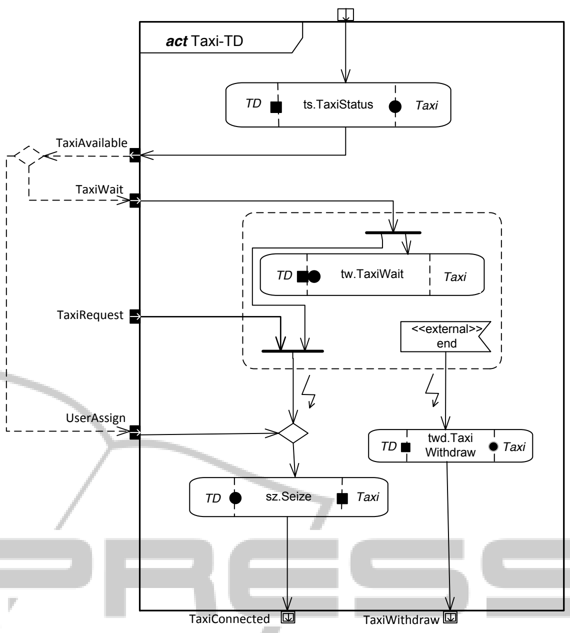
The *TaxiCentral* is not a trivial example since it has to deal with: (1) the multiplicity of taxis and users; (2) the queues and other data needed in order to coordinate users and taxis; (3) all the possible ways that events generated by taxis and users may interleave.

3 FUNCTIONAL SPECIFICATION

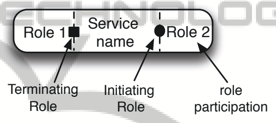
We consider a service here as a collaborative behaviour that may involve several components and more than one interface. In our method, a service provided by a distributed reactive system is specified in four steps: (1) the service structure is defined using UML collaboration diagrams; (2) the interface functionality is defined using UML activity diagram; (3) the core functionality is defined using UML activity diagram; (4) the interface and core functionality is connected. Each of these steps are illustrated in this section using the *TaxiCentral*.



(a) *User-TD* Interface Functionality Model.



(b) *Taxi-TD* Interface Functionality Model.



(c) Notation for *CallBehaviourAction*.

Figure 3: Interface Functionality Models of *TaxiCentral*.

3.1 Service Structure

The method guidelines for creating the service structure (SS) are described below:

SS1: *Use UML collaboration diagram to define the structure of a service.*

SS2: *Identify the participants and interfaces of the service from the functional requirements document. Represent the participants as roles in the collaboration diagram.*

SS3: *Decompose the service behaviour into collaboration-uses, where each collaboration-use encapsulates the interactions needed to carry a domain event mentioned in the requirements.*

SS4: *Bind the roles of each collaboration-use to the roles of the main collaboration.*

SS5: *Indicate the initiating role of each collaboration-use with a filled circle and terminating role with a filled square*[a].

[a]Filled circles and squares are not standard UML notations, but can be provided by profiling. They represent information needed during subsequent behaviour analysis and design synthesis.

SS6: *Add references to activity diagrams that will define the behaviours of interfaces identified in 'SS2' (detailed in Section 3.2).*

By inspecting the functional requirements given in Section 2, three main roles are identified in the *TaxiCentral*: the *User*, the *Taxi* and the *TD* (*Taxi-Dispatcher*). Likewise, three interfaces are identified namely: *User-TD*, *Taxi-TD* and *User-Taxi*. The UML collaboration diagram of *TaxiCentral* is shown in Figure 2. Each collaboration-use, for instance *TaxiReq*, *UserWait*, etc. encapsulates interactions needed to carry the events of the *TaxiCentral*. The decomposition of collaborations into roles and collaboration uses is one important step towards mastering complexity. Each elementary[1] collaboration involves only two roles and has a limited complexity making a complete definition of its behaviour practically feasible.

3.2 Interface Functionality

The method guidelines for interface functionality (IF) are as follows:

[1]We consider collaborations that are not further decomposed as elementary collaborations.

IF1: *Develop activity diagrams of each interface identified in 'SS2' and referenced in 'SS6'.*

IF2: *Map the domain events, related to an interface, from the requirements to pins*[b] *on the boundary of that interface activity. The name of a pin should correspond to a particular event.*

IF3: *Add a 'CallBehaviourAction' for each collaboration-use performed on the interface.*

IF4: *Add partitions on each 'CallBehaviourAction' to indicate the participating roles. Mark the initiating and terminating roles with filled circles and squares, respectively.*

IF5: *Make flows and activity nodes to define the ordering of the CallBehaviourActions identified in 'IF3' that can be enforced locally in the interface. This may include interrupting regions and interrupting flows, forks, joins merges, choices and other elements of UML activities.*

IF6: *For the ordering that is enforced externally, make flows to and from the corresponding pins. In the case of data dependency, connect the pins externally with dashed flows representing the external ordering required.*

[b]The activity diagrams can have all types of pins that UML allows, i.e. initiating, terminating, streaming and alternative pins.

The resulting interface functionality models for the *TaxiCentral* are depicted in Figure 3. Note that an interruptible region and interrupting flows have been used to specify that the *UserWait* collaboration-use shall be terminated when a *Taxi* becomes available or the *User* ends. The interruptible activity region is depicted by a dashed rectangle.

The dashed flows in Figure 3 represent essentially how the responses to incoming events shall be ordered by the core functionality depending on its data. Hence they represent what we may call a *data dependency*. In addition to *data dependency* there may be *event dependency* when events on one interface trigger actions on another interface. In the *TaxiCentral*, the arrival of an available *Taxi* on the *Taxi-TD* interface shall lead to actions on the *User-TD* interface, taking the user out of queue and assigning the *Taxi* if there is a waiting *User*. This case of *event dependency* is taken care of by the un-connected input pin *TaxiAvailable* on the *User-TD* interface. A corresponding event therefore has to be generated by the core logic.

Development of interface behaviours separately is much simpler and more manageable than to develop a complete behaviour directly as in the integrated approach. Each of the two interfaces presented in Figure 3 is relatively simple and easy to understand even

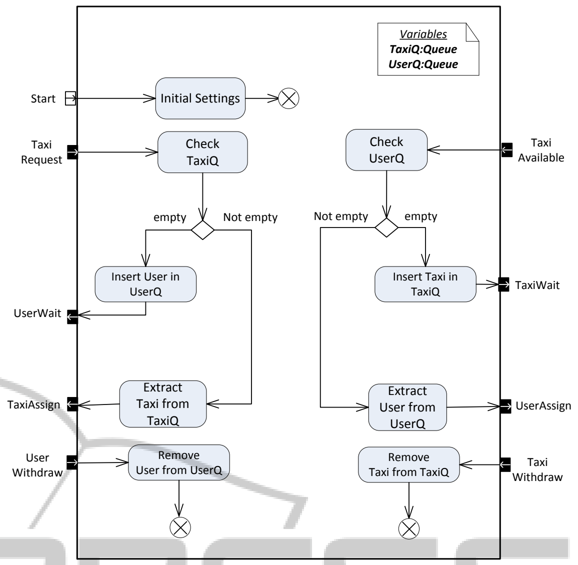


Figure 4: The Core Functionality Model of *TaxiCentral*.

for a non-technical person (with a little explanation) and therefore useful for discussions and validation with end users. Moreover, they define a behaviour contract that both parts of the interface must obey.

3.3 Modelling the Core Functionality

The method rules for core functionality (CF) are as follows:

CF1: *Define the activity that contains the core functionality by using a UML activity diagram.*

CF2: *Map the domain events from the requirements to streaming pins on the edge of the activity that defines the core functionality. The name of each pin should correspond to the name of an event defined in the requirements.*

CF3: *Define local variables in the activity that can hold the data it shall handle according to the requirements.*

CF4: *For each domain event to be handled (represented by a streaming pin) define an internal activity flow that performs the actions given in the corresponding requirement.*

CF5: *If necessary add pins to start and stop the activity.*

The resulting core functionality of the *TaxiCentral* is illustrated in Figure 4. The functional requirements of the *TaxiCentral* in Section 2 tells that it's core functionality needs to maintain a user queue (*UserQ*) and a taxi queue (*TaxiQ*) to coordinate multiple users and taxis. The variables are indicated by a note on the upper right corner of the activity diagram as shown in

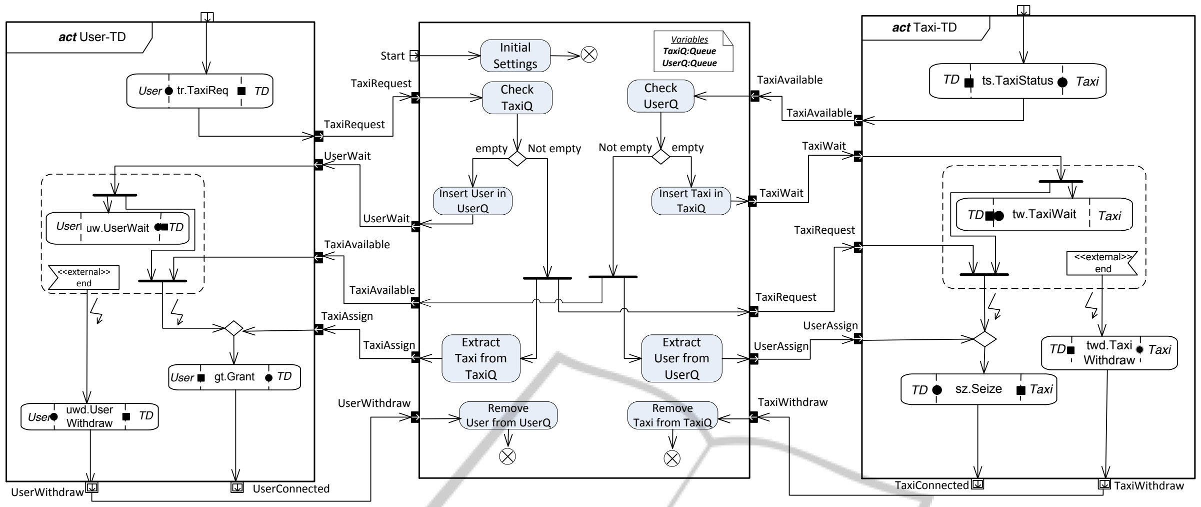


Figure 5: Composition of core functionality model and interface functionality models resulting in global behaviour model of the *TaxiCentral*.

Figure 4 since UML does not have any specific notation to represent variables in activity diagrams.

Although this model may appear to be very internal, it actually specifies the core behaviour which is of importance for end-users, and therefore relevant in a specification. In purely interface oriented specifications this functionality is completely missing.

3.4 Composing the Core and Interface Functionality

Once the interface functionality and core functionality models are defined, we need to ensure their correct alignment and coordination in order to compose the complete service behaviour. The method rules for composing the complete service behaviour (CSB) are as follows:

CSB1: *Find matching pins on the core functionality model and the interface functionality models. Connect the matching pins.*

CSB2: *If a matching output pin cannot be found in the core functionality, look into the core functionality model to identify the appropriate flow where the required event is represented. Add a fork to the flow identified and an output pin. Connect the fork to the pin.*

The resulting complete behaviour of the *TaxiCentral* is shown in Figure 5. It also illustrates the rule ‘CSB2’ where the matching *TaxiAvailable* and *TaxiRequest* output pins are not found originally on the core functionality model. They are then added to the core functionality model and connected to the newly added fork on the corresponding flow.

Note that the interface functionality and core functionality are defined in separate modules, that are partly independent and may be modified separately as long as their interface remains unchanged. This is one of the advantages of the interface-modular approach.

4 METHOD VALIDATION

First we comment that the development of a global interface-modular specification may not be quite as straight forward as suggested in the presentation above. In practice one may need to iterate in order to discover and resolve all dependencies and to develop the interface and core functionalities to the level presented in Figure 5.

We may now ask about the generality of the approach and completeness of the resulting global specification.

For the core functionality, activity diagrams are both well suited and general enough. They can easily be expressed in a tool like Reactive Blocks (Kraemer et al., 2009; ReactiveBlocks, 2014) that will analyze the model and generate corresponding code. For interface functionality, the challenge is to cover all possible event orderings. This is achieved by the interface focus, the decomposition into elementary collaboration-uses and the ordering constructs of activity diagram. The functionality of the *TaxiCentral* has been implemented in several versions using the Reactive Blocks tool (Kraemer et al., 2009; ReactiveBlocks, 2014). This includes both the core functionality and the interface functionality including remote communications using a variety of protocols. A lesson from this is that our use of activity diagrams is quite general and enable sufficient completeness for design synthesis. We have tested our approach on a variety of other problems too, in order to uncover possible shortcomings. Among these is the *bCMS* system (bCMS, 2012) which is more comprehensive and more distributed than the *TaxiCentral*. The *bCMS* core functionality needs to be partitioned and distributed accordingly. Other systems on which we have tested our approach are: a *LiftSystem*; an *Access-ControlSystem*; and a *Tele-MedicineSystem*. From this we have encouraging indications that the approach is sound. These examples are not explained in this paper due to lack of space.

5 RELATED WORK

Expressing the core functionality using activity diagrams is similar to the common practice in work-flow modelling used in the business process development. Factoring out interfaces in separate interface contracts is also according to common practice in the same domain as seen in web services and SOA. What is not so common is to define the two aspects in modules that can be directly composed using pins. Within the embedded and reactive systems domain, it is common to specify interface functionality using sequence diagrams, but less common to specify core functionalities separately. It is more common to map (incomplete) sequence diagrams to state machines that integrate interface and core functionality during (manual) design.

In the literature, one can find many proposals to represent high-level service specification such as sequence diagrams and Use Case Maps (UCM) (Buhr, 1998; Castejón, 2005). Interactions, for instance UML sequence diagram, are commonly used to express collaborative behaviour using messages which are exchanged between components. But sequence diagrams can normally be used for limited scenarios only and contain other drawbacks mentioned in (Castejón, 2008), (Zaha et al., 2008).

The WSCI specification (Arkin et al., 2002) describes the interface of a web service participating in a choreographed interaction using XML-based language. Unlike our approach, a WSCI interface describes only one partner's participation. The WSCI does not describe how a web service manages message ordering which in our approach is explicitly handled. Other approaches have been proposed (Beyer et al., 2005a; Beyer et al., 2005b; Mencl, 2004) to define interface behaviours, but none of them models interface behaviour as activities ordering elementary collaborations. The problem addressed in the multi-view point approach proposed in (Dijkman and Dumas, 2004) is similar to ours. But, there are important differences that sets our work apart from their approach. Interface behaviours for instance are one-sided in their approach. Moreover, their approach does not factor out internal tasks as we do in the core functionality.

Deriving interface contracts by projection from complete component behaviours have been much studied and several approaches exist (Bræk and Haugen, 1993; Floch and Bræk, 2003; Sanders et al., 2005). However, the opposite problem of specifying interfaces first, and then composing the complete component behaviours has not been so well researched in the past.

The interface behaviours we develop, define the behaviour that is visible on each particular interface. This may be considered as a *projection* of the complete behaviour of a component or a system. One of the original methods of projections is proposed in (Lam and Shankar, 1984) to reduce the complexity of analyzing non-trivial communication protocols. The method breaks up the protocol analysis problem into smaller problems by constructing a smaller image protocol system using refinement algorithms that preserves properties of the original protocol system. Our method is inspired by similar reduction of complexity by constructing smaller interface behaviours as projections. In doing this we seek to precisely define the visible interface behaviour that users and other entities in the environment will observe, while hiding other interfaces and details of the core functionality.

Various mathematical approaches have been proposed to define choreography semantics for example Labelled Transition System (Salaün and Bultan, 2012). Activity traces are used by (Qiu et al., 2007) to represent choreography. Most of these proposals lead to manual synthesis of the components. Whereas, once the system is specified using our interface-modular approach, synthesis of the components can be automated by using the rules published in (Fatima and Bræk, 2013).

6 CONCLUSION AND FUTURE WORK

We have presented an interface-modular method for system specification which is suitable to become the source of highly automated design synthesis. The method is demonstrated by a non-trivial case study, the *TaxiCentral*. Our method approaches the complexity of global behaviour specification problem by factoring out two separate aspects of a system i.e. ‘core functionality’ and ‘interface functionalities’. Interface behaviours are decomposed using collaborations and defined as activities ordering elementary collaborations expressed using activity diagrams. This separation allows interfaces and core functionality to be combined in different ways as long as their internal interfaces are unchanged.

To our knowledge, the interface-modular method we have presented is original in the way it defines interface behaviour, separates the interface behaviour and core functionality, and composes interfaces and core functionality.

Currently, we are working on using the interface contracts (which is a by-product of our method) for compositional validation to ensure correct interworking and dynamic binding.

REFERENCES

Arkin, A., Askary, S., Fordin, S., Jekeli, W., Kawaguchi, K., Orchard, D., Pogliani, S., Riemer, K., Struble, S., Takacsi-Nagy, P., et al. (2002). Web service choreography interface (wsci) 1.0. *Standards proposal by BEA Systems, Intalio, SAP, and Sun Microsystems*.

bCMS (2012). bcms case study document. http://cserg0.site.uottawa.ca/cma2013re/CaseStudy.pdf. Accessed On: 22-09-2014.

Beyer, D., Chakrabarti, A., and Henzinger, T. A. (2005a). An interface formalism for web services. In *Proceeding of the First International Workshop on Foundations of Interface Technologies*.

Beyer, D., Chakrabarti, A., and Henzinger, T. A. (2005b). Web service interfaces. In *Proceedings of the 14th International Conference on World Wide Web*. ACM, New York.

Bræk, R. and Haugen, Ø. (1993). *Engineering Real Time Systems*. Prentice Hall.

Buhr, R. J. A. (1998). Use case maps as architectural entities for complex systems. In *IEEE Transactions on Software Engineering*, volume 24(12). IEEE Press.

Castejón, H. N. (2005). Synthesizing state-machine behaviour from uml collaborations and use case maps. In *SDL 2005 Model Driven*, volume 3530. LNCS, Springer.

Castejón, H. N. (2008). *Collaborations in Service Engineering: Modeling, Analysis and Execution*. PhD Thesis, Department of Telematics, NTNU.

Castejón, H. N., Bræk, R., and Bochmann, G. V. (2007). Realizability of collaboration-based service specifications. In *Proceedings of the 14th Asia-Pacific Soft. Eng. Conf. (APSEC07)*. IEEE Computer Society Press.

Dijkman, R. and Dumas, M. (2004). Service-oriented design: A multi-viewpoint approach. In *International Journal of Cooperative Information Systems*, volume 13.

Fatima, U. and Bræk, R. (2013). On deriving detailed component design from high-level service specification. In *System Analysis and Modeling About Models*, volume 7744. LNCS, Springer.

Floch, J. and Bræk, R. (2003). Using projections for the detection of anomalous behaviours. In *SDL 2003 System Design*, volume 2708. LNCS, Springer.

Kathayat, S. B. and Bræk, R. (2011). Analyzing realizability of choreographies using initiating and responding flows. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa, pages 6:1–6:8, New York, NY, USA. ACM.

Kathayat, S. B., Le, H., and Bræk, R. (2011). A model-driven framework for component-based development. In *SDL 2011 - Integrating System and Software Modeling*, volume 7083. LNCS, Springer.

Kraemer, F. A., Slåtten, V., and Herrmann, P. (2009). Tool support for the rapid composition, analysis and implementation of reactive services. In *Journal of Systems and Software*, volume 82(12). Elsevier.

Lam, S. S. and Shankar, A. U. (1984). Protocol verification via projections. *IEEE Transactions on Software Engineering*, 10(4).

Mencl, V. (2004). Specifying component behavior with port state machines. In *Electronic Notes on Theoretical Computer Science*.

Qiu, Z., Zhao, X., Cai, C., and Yang, H. (2007). Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 973–982, New York, NY, USA. ACM.

ReactiveBlocks (2014). Reactive blocks - the tool for professional java developers. http://www. bitreactive.com. Accessed On: 10-12-2014.

Salaün, G. and Bultan, T. (2012). Realizability of choreographies using process algebra encodings. *Services Computing, IEEE Transactions on*, 5(3):290–304.

Sanders, R. T., Bræk, R., Bochmann, G. V., and Amyot, D. (2005). Service discovery and component reuse with semantic interfaces. In *SDL 2005 Model Driven*, volume 3530. LNCS, Springer.

Zaha, M. J., Dumas, M., Hofstede, A. H. M., Barros, A., and Decker, G. (2008). Bridging global and local models of service-oriented systems. In *IEEE Transactions on Systems, Man., and Cybernetics*, volume 38(3). IEEE Press.