Description of Protocol Rules

Andreas Speck, Sören Witt, Sven Feja

[aspe|swi|svfe]@informatik.uni-kiel.de

Abstract: Protocols are formal models which may be used to define the interactions within processes. Moreover, protocols may be the base of process improvement e.g. by applying game theory.

In the paper we focus on models of interactions between system components (which we name *versions*) and the possibilities to verify that the version systems fulfill the rules (specifications) of the protocols. First, we consider the static relationships between the versions as base. Second, we discuss the dynamic interactions between these versions and present a model checking-based approach to verify the interaction specifications of the protocols.

Such verified interaction sequences (or processes) are a starting point of optimizations by game theory approaches.

1 Introduction

Protocols allow describing the interactions between different entities. In the software development such objects may be classes / objects, components (versions of components) or services which are cooperating. Such a cooperation or the set of interactions may result in a process, e.g. a business process. There are numerous approaches to optimize such processes. Game theory for instance helps evaluating different paths in the process models. Due to the given criteria an optimal solution may be found.

The question before applying optimization techniques is the correctness of the different interactions and the processes or games. In this paper we present two basic issues to be checked before an evaluation of the process:

1. Are all objects considered in the process? Are the static relations between these objects correct?
2. Are the interactions between the objects in a correct order? Does the interaction follow the specifications in the protocol?

In this paper we present techniques which allow expressing rules which represent relationships between the objects and the order of activities (process). The idea is to present a generic model which may be supported by different checking techniques. The checking techniques are intended to support games. Although we do not elaborate this in detail in the current paper.

erschienen im Tagungsband der INFORMATIK 2011
Lecture Notes in Informatics, Band P192
ISBN 978-3-88579-286-4

2 Model

In the paper we take the example of software development in order to use a comparatively generic model: Here we consider *Components* as building blocks of systems which may be combined in order to create a software system. Inside of the *Components* are *Objects* providing the requested functionality. The *Components* may be arranged hierarchically which means that *SuperComponents* consist of *SubComponents*. This model is very similar to the object oriented modeling concept. However it may also be mapped to web services or business process models like BPMN. Additionally we combine the component model with the *EventPort* concept of [LK99]. The *EventPort* model specifies the components' interfaces with *InPorts* and *OutPorts* which are finite state machines representing the protocol of the in and out communication of a component. The *InPorts* and *OutPorts* are based on the module interface concept [GP70].

Since incoming and outgoing communication is not sufficient to specify all properties of a component including their activity and the relationships between components we extended the *EventPort* model by adding states defining the internal behavior of a component. The extended model consists of the component's input (represented by *InPorts*), its output (*OutPorts*) and its internal behavior which represents the connections between the *InPorts* and *OutPorts* (if an incoming message triggers an outgoing message). Each component may have an arbitrary number of *InPorts* and *OutPorts*.

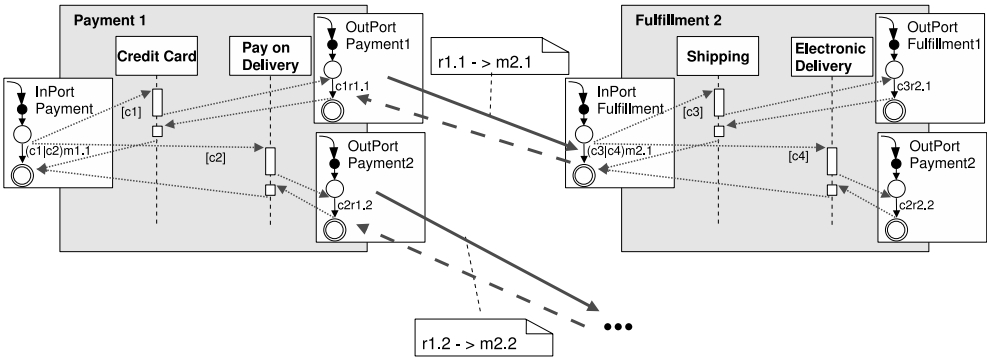

Figure 1: Static Relationship: *Component* Compositions

Figure 1 depicts a notation for the extended *EventPort Component* Model. *InPort* $Payment$ of component *Payment 1* first accepts message $m1.1$ when conditions $c1$ or $c2$ are true (Boolean OR). In case condition $c1$ is true the payment is by credit card and *OutPort* $Payment1$ will react with out-message $r1.1$ which is mapped to ingoing message $m2.1$ of *InPort* $Fulfillment$ of component *Fulfillment 2*. The component *Fulfillment 2* will either in case condition $c3$ is true process the functionality *Shipping* resulting in the outgoing message $r2.1$ of *OutPort* $Fulfillment1$ or if condition $c4$ is true in outgoing message $r2.2$ of *OutPort* $Fulfillment2$.

Like all outgoing messages $r1.2$ of *OutPort* $Payment2$ will be caught by another component symbolized in the figure by the three dots.

3 Static Relationships

The components may be combined as aggregations using the services provided by other components (cf. figure 1) or as compositions containing other sub-components. A super-component containing other sub-components may define specific *InPorts* and *OutPorts* which may be derived from the *InPorts* and *OutPorts* of the sub-components.

The hierarchical order of components may be controlled by compositions. Only in case the condition is true (which means that the specifications of the protocol are fulfilled) the relationship is correct. We call this kind of composition *versioning* like proposed in [**?**] or [**?**]. We do not focus on the meaning of administrating and storing of different pieces of software.

Composition conditions (representing protocols) are a concept to define different component versions within a system family [PSC01]. This leads to an inductive definition of the construction of components (and component systems respectively):

$$Version\, V_i^0 \;=\; true \;\wedge\; C_i^0$$

C_i^0 represents the condition that has to be true for one component version V_i^0 on level 0. Several conditions may be unified in one condition.

V_i^0 represents the lowest level and thus atomic components. It is obvious that the Boolean value $true$ might be omitted. However, the current form emphasizes the principle of the recursive construction.

A super-component is defined as a set of conditions (a unification of the particular conditions of a certain super-component and all conditions of the sub-components contained in the super-component). A condition is expressed as Boolean expression.

An example for a super-component may be the following (cf. figure **??**, please note that `V1.2` should be read as $V_2^{(1)}$.

$$V_1^{(2)} \;=\; V_1^{(1)} \wedge V_2^{(1)} \wedge (V_3^{(1)} \vee V_4^{(1)}) \wedge \neg\, C8 \quad \text{with:} \quad C8 \;=\; (\,V_4^{(1)})$$

We call this representation a *version* according to [PSC01]. As opposed to a simple understanding of a version in temporal dimension only a version should be considered on multiple dimensions (cf. Related Work).

4 Dynamic Interactions

In order to realize a protocol the dynamic interactions between the components have to be considered. Figure 2 depicts an order of interactions between components / versions or to be more precise an order of interactions of sub-versions within a super-version.

The model defines a sequence of interactions. These may be considered as a process model. The question now is, if this process model will fulfill the specification in an interaction protocol.

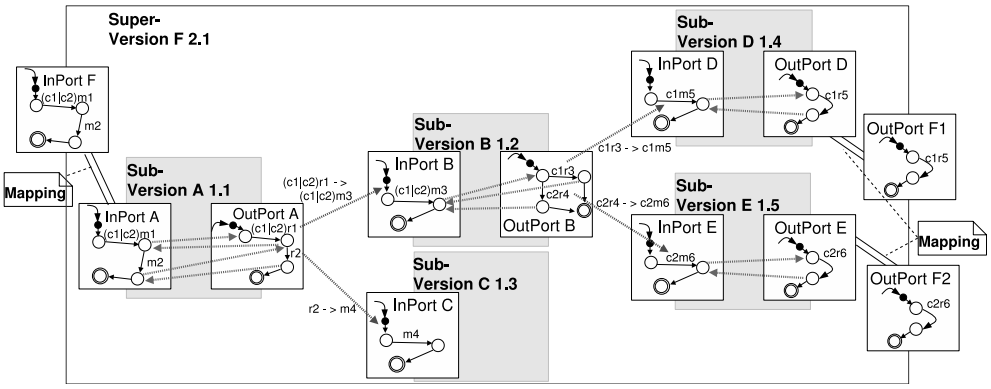

Figure 2: Modeling *Component* Hierarchies: *Versions*

The sequences of interactions like in figure 2 may be represented in automates. Such automate are then the base of the automated verification of protocol specifications. Model checkers may be used to verify the protocol specifications. Model checkers require a specific formalism to express the specifications or rules. This is the temporal logic – an extension of the Boolean logic with operators representing temporal relationships. In this paper we consider CTL (Computational Tree Logic) as temporal logic.

In our example such CTL protocol rules may be like:

- In case message $m1$ is received there should eventually be the resulting outgoing message $r5$. In CTL: `m1 -> EF r5`.
- Defining a specific order of messages:
  `(EF m5) -> A[¬ m4 U (m5 ∨ AF (¬ m5)]`. This means that if there exists eventually the message $m5$ then it has to be true that $m4$ may not be received until message $m5$ may not be received any more in the process.

  This specification is fulfilled due to the sequence in *OutPort A*: In this outport first $r1$ is send which results potentially $m5$. Only when the path $m5$ is finished then $r2$ is sent resulting $m4$.
  In other words: the specification is fulfilled.
- An example of a rule expressing the exclusiveness is:
  `(r5 -> ¬ AG r6 ) ∧ (r6 -> ¬ AG r5 )`. This means that either the message $r5$ is activated or the message $r6$ is excluded and vice versa.

Based on the verification by model checkers games (which may be alternative paths) may be evaluated. Approaches to realize this are for instance the connection of the paths with costs or any other value which may then aggregated by valued model checkers. The aggregated value is then the result of one specific game which has to be compared with other games. A similar approach may be applying probabilistic model checking. In this case the values or cost are not static but probabilities.

5 Related Work

EventPort automates [LK99] are used to describe and check processes according to protocol specifications. Further approaches applying port models are [DJ09] or [SPJF03]. The latter is our earlier work. [DJ09] introduces a language to describe components and component composition. This language aims to check component composition in three aspects: signature constraints, behavior compatibility and run time errors. In this paper we demonstrate how to transform such languages to checkable models. Another approach [TM08] presents verification focusing the interfaces between components. Here the internal behavior of the components is not considered. However, the interface interactions are more elaborated and support the variability of activities.

An aspect not yet discussed is that the application of a checking tool such as model checkers requires transformation mechanisms to convert the models and specifications in the appropriate formats. An example of such a powerful transformation concept is [FP07].

Model checking in general may support the evaluation of games. An example of the combination of model checking and games is [TvdHW08]. This approach applies a specific hybrid model checker. It allows calculating a strategic equilibrium. This concept may support the development of a generic concept to check different kinds of entities. An example of applying probabilistic checking is [BFW06]. In this example different strategies of agents may evaluated in detail. Applying probabilistic checking would be of interest when there are a large number of alternating paths with probabilities. This approach extends the pure comparison of games by introducing probabilities.

Up to now we only consider the checking based on automates with only one type of elements. Considering different types of elements as proposed in [Pul09] would support to evaluate different dimensions in a game. Examples for such different elements may be the various element types in ARIS EPC (*Event-driven Process Chains*) which may be considered in separate dimensions, e.g. a dimension of organizational flow or dimensions of functional flow. These different dimensions may reflect certain types of conditions.

6 Conclusion and Future Work

Protocols allow specifying rules for static relationships between software components (versions) which may also be objects or services. Moreover the dynamic interactions between the components (or other kind of software subsystems) may be specified. If there are means to specify the correct relations and interaction sequences then there is a base for optimization, e.g. by games.

In the paper we first focus on the verification of the relationships and dynamic sequences. We use *EventPorts* to model the interfaces between components. Correct compositions are represented by versions which are purely Boolean logic. The protocol specifications of these interactions are formulated in temporal logic. These may be verified by model checkers.

The pure model checking may be extended to valued or probabilistic checking which allows to aggregate values of the different paths representing different games. This is an base for applying game theory approaches to identify optimal solutions for systems.

References

[BFW06] Paolo Ballarini, Michael Fisher, and Michael J. Wooldridge. Automated Game Analysis via Probabilistic Model Checking: a case study. volume 149, pages 125–137, 2006.

[DJ09] Zuohua Ding and Mingyue Jiang. Modelling and Verification of Port Based Component Compositions. In *9th International Conference on Quality Software, QSIC '09*, pages 86–91. IEEE Computer Cociety, 2009.

[FP07] Daniel Fötsch and Elke Pulvermüller. Constructing higher-level Transformation Languages based on XML. In *Proceeding of the 6th International Conference on Software Methodologies, Tools and Techniques (SOMET'07), 7-9 November 2007, Rome, Italy*, pages 269–284, Rome, Italy, November 2007. IOS Press.

[GP70] Richard Gouthier and Stephen Pont. *Designing Systems Programs*. Prentice Hall, Englewood Clifs, 1970.

[LK99] Anthony Lauder and Stuart Kent. EventPorts: preventing legacy componentware. In *Proceedings of 3rd International Enterprise Distributed Object Computing Conference (EDOC 99)*, pages 224–232. IEEE Publishing, 1999.

[PSC01] Elke Pulvermüller, Andreas Speck, and James O. Coplien. A Version Model for Aspect Dependency Management. In *Proceedings of International Symposium of Generative and Component-based Software Engineering (GCSE 2001)*, LNCS 1241, pages 70 – 79. Springer, 2001.

[Pul09] Elke Pulvermüller. Reducing the Gap between Verification Models and Software Development Models. In *SoMeT*, volume 199 of *Frontiers in Artificial Intelligence and Applications*, pages 297–313. IOS Press, 2009.

[SPJF03] Andreas Speck, Elke Pulvermüller, Michael Jerger, and Bogdan Franczyk. Component Composition Validation. *International Journal of Applied Mathematics and Computer Science*, 12(4):581–589, January 2003.

[TM08] Mircea Trofin and John Murphy. Static verification of component composition in contextual composition frameworks. *International Journal on Software Tools for Technology Transfer (STTT)*, 10(3):247–261, 2008.

[TvdHW08] Nicolas Troquard, Wiebe van der Hoek, and Michael Wooldridge. Model Checking Strategic Equilibria. In *Model Checking and Artificial Intelligence, 5th International Workshop, MoChArt 2008, Patras, Greece, July 21, 2008. Revised Selected and Invited Papers*, volume 5348, pages 166–188. Springer LNCS, 2008.

[ZS97] Andreas Zeller and Gregor Snelting. Unified Versioning through Feature Logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398 – 441, 1997.

ISBN 978-3-88579-286-4