Verifying the Correctness of Component-Based Applications that Support Business Processes

Remco M. Dijkman João Paulo Andrade Almeida Dick A.C. Quartel
CTIT, University of Twente
{dijkman|almeida|quartel}@cs.utwente.nl

Abstract

Developing applications that properly support the enterprise is a difficult task. Failing to perform this task results in applications that are not accepted by the end-users and that frustrate daily conduct of business. In this paper we introduce a formal yet practical method that helps to design component-based applications that properly support the enterprise. The method can be used to verify whether the behavior of an application conforms to the behavior of the enterprise, where the behavior of the enterprise is specified in the form of business processes. The method helps to avoid applications being designed that support the enterprise in an incorrect manner.

1. Introduction

Applications should properly support the enterprise in which they are used. We claim that this is achieved if the joint behavior of applications and their users is equivalent to the intended behavior of the enterprise. This form of equivalence is called conformance. In this paper we show a formal yet practical method that helps to achieve conformance.

The method assumes that the intended behavior of the enterprise is described in the form of business processes. Business processes describe the tasks and the conditions under which tasks must be performed in order to achieve certain goals.

The method also assumes that the applications that support the enterprise are constructed by assembling components. The components we use typically correspond to concepts that have direct meaning in the enterprise, such as 'client' or 'account'. We call these components enterprise components. In order to guarantee that a business process is supported correctly by an assembly of enterprise components, the behavior of that assembly should be verified against this business processes. Since a business process defines constraints on the execution of business tasks, typical questions about component assemblies that can be answered by verifying them against a business process, are related to the enforcement of business constraints. Examples of such constraints are that *the client receives a requested item within 14 days after he/she ordered it* and that *if an item is received by the client, payment always follows*.

The method we propose uses a generic modeling technique that can be used to design both business processes and components [5,13]. This modeling technique has a precise formal semantics [11] that we can use to define an algorithm for verifying conformance.

The rest of this paper is organized as follows. Section 2 presents an outline of the method and some requirements on the design trajectory. Section 3 describes the theoretical underpinnings of the method and the procedure for conformance verification. Section 4 illustrates the application of the method with an example and section 5 presents some conclusions and future work.

2. Outline of the method

Our method assumes that applications are designed using stepwise refinement. Stepwise refinement starts out with a design that represents a rough outline of the structure of the system in terms of its parts (sometimes already called components at this level), the way in which these parts are interconnected and the behavior of the parts. Subsequently each of the parts is designed in more detail (refined) by splitting it up into multiple interconnected parts and/or by detailing its behavior. We call a design before a refinement step an abstract design, and a design after a refinement step a concrete design. In stepwise refinement, refinement of system parts can be applied repeatedly. Hence, a concrete design of one refinement step may be the abstract design in another refinement step. Refinement is repeated until a level of detail is reached that the designer considers suitable to start implementing onto component technology. A process of stepwise refinement is shown in figure 1. A design approach that uses stepwise refinement is, for example, Catalysis [4].

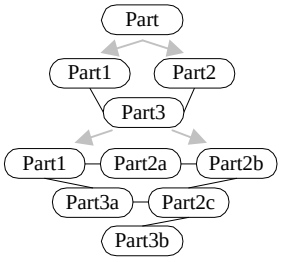


Figure 1. Stepwise refinement.

If we use business processes as a starting point for stepwise refinement, we may consider the entire enterprise including the application that supports the enterprise as a singular system (part). The behavior of this system is defined by the business processes and, consequently, business tasks correspond to activities that are performed by this singular system. The system

that represents the entire enterprise is refined by splitting it up into a part that represents the application under development and parts that form the environment of the application, such as, e.g., users of the application.

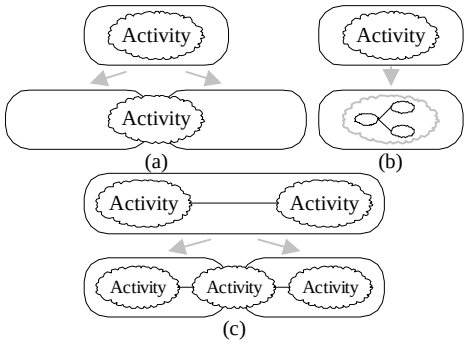


Figure 2. The refinement operators.

The behavior of a system may be refined by the following refinement operators:

1. An activity may be detailed by identifying individual contributions of system parts that perform the activity in cooperation (see figure 2a). The information that was produced, removed or updated by the original activity is now produced, removed or updated jointly by the contributing parts. As an example consider the activity 'record client data' that may be performed jointly by the system and a user who enters the information;
2. An activity may be detailed by any number of (related) sub-activities performed by the same system parts that performed the original activity (see figure 2b). The information that was produced, removed or updated by the original activity is now produced, removed or updated by the sub-activities together. As an example consider the activity 'record client data' that may be detailed by the sub-activities 'record client name' and 'record client address';
3. An activity has to be added if communication is required between two system parts to enforce the relation between two system parts (see figure 2c). As an example consider that the activity 'enter client data' and 'verify client data' are performed by different parts. Then a shared activity is necessary to carry the client data from one part to another.

Having identified the ways in which an abstract design may be refined into a concrete design, we distinguish two approaches to ensure that a component assembly correctly supports the original business processes: (i) by ensuring that designers only use the aforementioned refinement operations, in which case the refinement may be considered conformant by construction, and; (ii) by checking after a design step whether the concrete design can be reached by applying the refinement operations. Since a designer may experience the refinement rules as overly restrictive, in particular because refinement is a creative activity, we opt for the latter approach (ii).

However, it is not feasible to verify conformance of a concrete design to an abstract design by trying to get from the abstract design to the concrete design by applying the refinement operators. The reason for this is that the refinement operators may be applied in any way and in any combination. In contrast, we will show that there is only one way to reach an abstract design from a concrete design by applying inverted refinement operators.

Therefore, in this paper we will show the outline of an algorithm to verify the conformance of a component assembly to a business process by applying the inverted refinement operators to a component assembly. The inverted refinement operators are:

1. Remove the boundaries between the parts. Consequently, each concrete sub-activity that is performed jointly by two or more parts is now performed by a single part;
2. Integrate the concrete sub-activities that belong to the same abstract activity into one activity, and integrate the information produced, removed or updated by these sub-activities; and
3. Remove the concrete activities that are added because communication between two parts was required.

After we applied these inverted refinement operations we are left to check whether each of the resulting activities are related in the same way as the original activities, and whether the information produced, removed or updated by these activities is the same as that of the original activities.

3. Theory

In this paper, we use the Interaction Systems Description Language (ISDL) to verify the conformance of a component assembly to a business process. The reason for using ISDL is that it provides a rigorous formal semantics and algorithms to perform conformance verification. We realize that it is not realistic to assume that in each concrete situation an ISDL design of the business process and of the component assembly exists. Therefore, in future work, we will provide mappings from commonly used modeling techniques to ISDL an vice versa, such that these modeling techniques can benefit from ISDL's semantics and algorithms. In this section we explain ISDL and an algorithm for conformance verification.

3.1. The Interaction Systems Description Language

The ISDL consists of three basic concepts: action, interaction and causality condition.

An *action* represents the successful completion of some unit of activity performed by a single system part. An *interaction* represents the successful completion of a common activity performed by two (or more) system parts. An *interaction contribution* represents the participation of an individual system part in the interaction. An action is graphically represented as a circle. An interaction is graphically represented as a segmented circle, where each segment of the circle represents an interaction contribution.

The *information*, *time* and *location attributes* of an (inter)action represent the result established in the activity, the point in time at which this result is available and the location where the result is available, respectively. The information (*i*), time (*t*) and location (*l*) attributes are graphically represented within a text-box attached to the (inter)action. The result that is established in one (inter)action can be referred to by all subsequent (inter)actions. *i(name)* refers to the result established in the (inter)action with the corresponding name. An (inter)action is atomic at the level at which it is considered in the

sense that if an (inter)action occurs, the same result is established and made available at the same time moment and at the same location for all system parts involved in the activity. Otherwise, no result is established and no system part can refer to any intermediate results of the activity. Constraints can be defined on the possible outcomes of the values of *i*, *t* and *l*. In case of an interaction, each interaction contribution defines the constraints of the corresponding system part, such that the values of *i*, *t* and *l* must satisfy the constraints of all involved system parts, otherwise the interaction cannot happen.

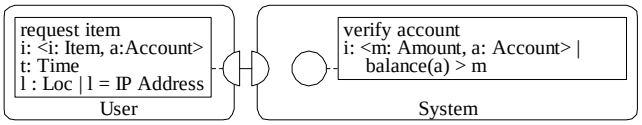


Figure 3. An example of an action and an interaction.

Figure 3 shows an example of an action and an interaction. The interaction represents the successful completion of a joint activity of a system and its user to request an item to buy. Two results are obtained in this activity: the item that the user wants to buy and the account from which the item can be paid. The completion of the activity occurs at some time moment *t*, on a location *l* that is constrained to be a certain IP address. The action represents the successful completion of an activity of the system to verify the user's account. The figure also shows how we can delimit the behavior of a system (part) by means of a *behavior block*.

A *causality condition* is associated with each action, or interaction contribution, describing the condition for this action or interaction contribution to happen, in terms of the occurrence of other (inter)actions. We distinguish between four basic causality conditions for the occurrence of some action or interaction contribution *a*:

- (inter)action *b* must happen before *a*. This is graphically represented as: ⓑ→ⓐ;
- (inter)action *b* must *not* happen before, nor simultaneously with *a*. This is graphically represented as: ⓑ⊣ⓐ;
- (inter)action *a* happens simultaneously with *b* (due to space limitations, we do not consider synchronization in this paper any further);
- (inter)action *a* is always enabled. This is graphically represented as: →ⓐ.

And- and or-operators can be used to define more complex causality conditions. The *and*- and *or*-operator are graphically expressed by the symbols ■ and □, respectively. Using the *and* operator we could, for example, express the causality condition: *a must happen before b and c must not happen before, nor simultaneously with b*. The causality condition for an interaction is implicitly defined by the *and* of the causality conditions of all its interaction contributions.

Figure 4 shows an example of a set of actions with causality conditions. The figure shows that *a* is always enabled, that *b* and *c* are enabled if *a* has happened and that *b* and *c* exclude each other, that *d* is enabled after *b* has happened and *e* is enabled after *c* has happened.

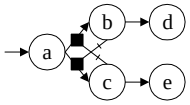


Figure 4. An example of a behavior design.

Due to space limitations our introduction of ISDL is rather short. For an explanation of more complex concepts, such as repetitive behavior and probability, and for a formal syntax and semantics we refer to [11].

3.2. Conformance Verification

In section 2 we explained the inverted refinement operators that are used for conformance verification. In this section we explain these operations for designs that have a semantics in ISDL.

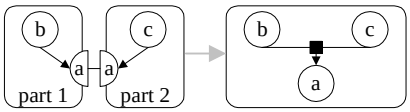


Figure 5. Removing boundaries between parts.

3.2.1. Remove Boundaries between Parts. The first step in conformance verification is to remove the boundaries between system parts that were introduced during a refinement step. When we remove the boundaries between parts, each interaction that happens between these parts can be substituted by an action. The causality condition of this action is formed by a causality condition that is the *and* of the causality conditions of each of the original interaction contributions. For example, interaction *a* between part 1 and part 2 in figure 5 is substituted by an action *a* with a causality condition that corresponds to the *and* of the causality condition of the contribution of part 1 to *a* and the causality condition of the contribution of part 2 to *a*.

After substituting an interaction by an action, we may have to simplify the causality condition of the resulting actions. Commonly needed simplification rules are: *x and always = x*, *x and x = x*, *x or x = x*.

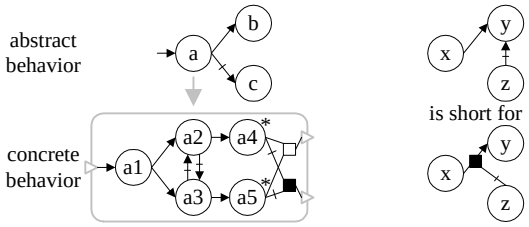


Figure 6. An example of refinement.

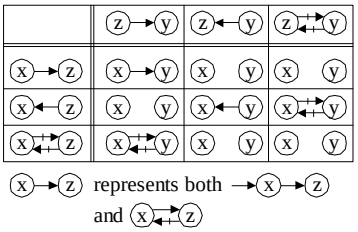
3.2.2. Integrate Concrete Actions. The second step in conformance verification is to integrate the concrete actions that were split up during a refinement step. To do this we assume that the relation between an abstract (inter)action and the concrete actions that refine it is known, either because the user performed the refinement step in a tool that stores this relation, or because the designer indicated the relation as part of the conformance verification. We show this relation by drawing a gray behavior block around the concrete actions that refine the same abstract (inter)action as shown in figure 6. A triangle pointing into the block (called *entry*) represents a condition that influences the occurrence of the abstract (inter)action. A triangle pointing out of the block (called *exit*) represents the influence of the abstract (inter)action on the causality condition of another abstract (inter)action. In figure 6 the entry represents the

causality condition of abstract action *a*. The exits represent the influence of *a* on the causality conditions of *b* and *c*.

We distinguish between two types of concrete actions in the blocks: final actions and inserted actions. Final actions are actions that represent the completion of an entire block, and therefore represent the completion of the abstract (inter)action. Final actions can be identified, because they contribute to the causality conditions of the exits. In figure 6 we marked the final actions with an asterisk. Inserted actions are actions that are not final actions. They are actions that are inserted with respect to the abstract (inter)action. We integrate concrete actions by removing inserted actions and replacing final actions by a single action.

Removing Inserted Actions. When we remove an inserted action *z*, the indirect causality conditions that run via *z* have to be preserved. For example, if we remove inserted action *a2* from figure 6, the indirect enabling condition from *a1* via *a2* to *a4* would have to be preserved. So after removing *a2* there should be an enabling condition from *a1* to *a4*. Table 1 shows how to preserve indirect causality conditions that run via inserted action *z*. It says that if *z* and *x* depend on each other as shown in row i and *z* and *y* depend on each other as shown in column j, then, after removing *z*, *x* and *y* depend on each other as shown in cell (i, j). If the dependency between *x*, *y*, and *z* is not shown in table 1, then there will be no relation between *x* and *y* after removing *z*.

Table 1. Removing indirect causality relations.



If multiple indirect causality conditions run via *z* (and neither of these contains an *or*) then we: (i) identify each combination *x*, *y* that *z* relates, (ii) remove *z* from each of these combinations according to table 1, and (iii) integrate the resulting combinations. An example of this is shown in figure 7, where we: (i) identify the combinations that *a2* relates, (ii) remove *a2* from each of these combinations, and (iii) integrate the combinations again.

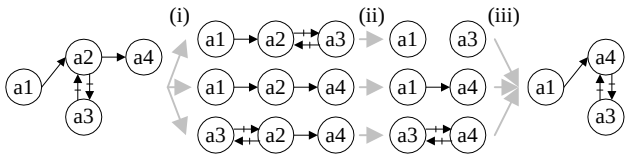


Figure 7. Removing multiple indirect causality relations.

If any of the indirect causality conditions that run via *z* contains an *or*, then we: (i) split up the design into the alternative cases, (ii) remove *z* from each of these cases according to table 1, and (iii) integrate the alternative cases as alternative causality conditions again. An example of this is shown in figure 8, where the indirect relations that run via *c* contain an *or*. In the figure we: (i) split up the design in the alternative cases '*a enables c, c enables d*' and '*b enables c, c enables d*', (ii) remove *c* from both of these cases, and (iii) integrate the cases as alternatives.

Hence, removing inserted actions transforms figure 9a into figure 9b. Removing *a2* and *a3* happens according to figure 7, and removing *a1* directly relates *a4* and *a5* to the *entry* according to cell (1,1) in table 1.

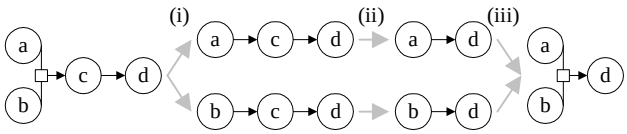


Figure 8. Removing indirect causality relations that contain an or.

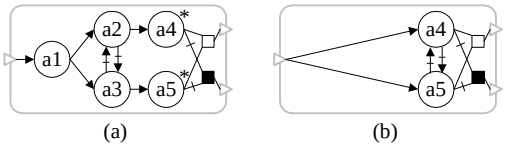


Figure 9. Removing inserted actions.

Replacing Final Actions. While final actions correspond to the completion of the original abstract action, we do not yet know the combination in which they correspond to the completion. For example, the completion of *a* final action may correspond to the completion of the abstract action, or the completion of *all* final actions may correspond to the completion of the abstract action.

We call the combination in which final actions correspond to the completion of the corresponding abstract action: the completion condition. Without proof, we claim that the completion condition of an abstract action is equal to the causality condition of an exit that corresponds to the enabling of another abstract action. For example, in figure 6 the completion condition of *a* is *a4 or a5*, because this is the condition of the exit that corresponds to the enabling of *b*.

When we replace final actions by a single action, the causality condition of this single action must be the completion condition of the corresponding abstract action. Each final action that appears in the completion condition is replaced by the causality condition of this final action. Finally, the causality conditions of exits that represent abstract enabling conditions are replaced by enabling conditions from the integrated action to the exit. The causality conditions of exits that represent abstract disabling conditions are replaced by disabling conditions from the integrated action to the exit (provided they are the inverse of causality conditions of exits that represent abstract enabling conditions. The inverse in calculated by changing enabling relations to disabling relations and *and*s to *or*s and vice versa).

Hence, replacing final actions by a single action, transforms figure 10a into figure 10b. The transformation is achieved by replacing final actions *a4* and *a5* by *a*. The completion condition is the causality condition of the topmost exit (*a4 or a5*) where *a4* is replaced by its causality condition (*entry*) and similarly *a5* is replaced by *entry*. Hence, the causality condition of *a* is *entry or entry*. The causality condition of the topmost exit becomes the enabling of *a*, and the causality condition of the bottom exit becomes the disabling of *a*.

After integrating the concrete actions, we may again need to simplify the design. After simplification figure 10b is transformed into figure 10c.

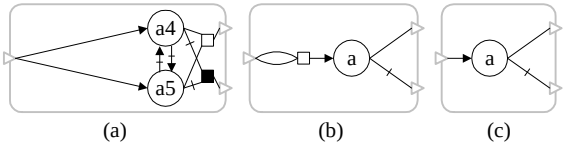


Figure 10. Integrating final actions.

3.2.3. Remove Interactions Used for Communication. The third step in conformance verification is to remove the interactions that were inserted to make different parts communicate. We do this by integrating these interactions into actions according to 3.2.1, and removing them in the same way as we remove inserted actions according to

3.2.4. Verify Conformance. After these steps are performed, the resulting actions and their relations should correspond in a one-to-one fashion to the original design. A one-to-one correspondence between abstract and concrete design is called strong conformance. Alternatively, we may use weak conformance verification approaches [11] to allow more freedom in the implementation.

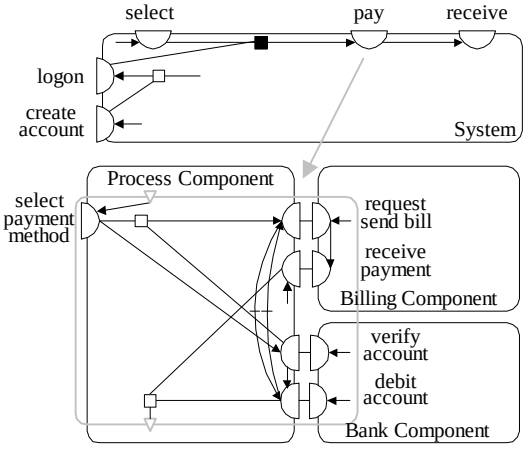


Figure 11. An example of refinement in an application.

4. Example

A simple example of the application of the method described above is shown in figure 11.

Figure 11 shows a business process that a user experiences when he/she buys something online. The business process has already been split-up in a user part and an application part. The figure shows the application part. The business process shows that the user can always logon using an existing account, select items to buy, or create a new account. The user can also logon after he or she has created a new account. Payment is only enabled after the user has both selected an item to buy and has logged on. Reception of goods happens after the user has paid.

Figure 11 also shows a refinement of the payment interaction. The gray behavior block represents the original interaction 'pay' and the entry and exit of the block represent the enabling of 'pay' by 'select' and 'logon' and the influence of the payment interaction on the enabling of 'receive' respectively.

As the figure shows, the original payment interaction is refined by five interactions. Initially, the user will be allowed to select a method of payment. Depending on the selected method of payment (this dependency is not shown, because in this paper we do not discuss conditions that are based on data) a 'send bill request' is issued to the billing component or we verify whether the user has enough money in the bank on the bank component. If the user does not have enough money in the bank, then he or she also receives a bill. Hence, 'request bill' is enabled by either 'verify account' or directly by 'select payment method'. If the user has enough money the account will be debited. Either debiting the users account directly or receiving a notification that the payment has happened from the billing component concludes the abstract 'pay' interaction.

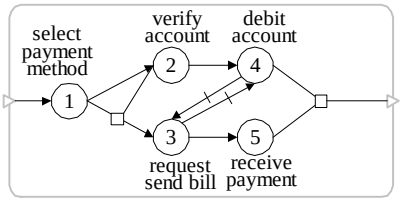


Figure 12. An integrated concrete design.

To verify whether the refinement conforms to the original design, we first remove the boundaries between the components. Removing the boundaries between components yields figure 12. From figure 12 we can conclude that 4 ('debit account') and 5 ('receive payment') are final actions and hence 1 ('select payment method'), 2 ('verify account'), and 3 ('request send bill') are inserted actions. Figure 13 shows how to remove inserted action 3. First we identify the indirect causality conditions that run via 3. Then we split up the design into two alternative cases. Each of these cases can be split up into three indirect causality conditions. These causality conditions can be removed according to table 1. Now the design can be integrated again, first into the two alternative cases and then into the structure with 3 removed. Further removing 1 and 2 yields figure 14.

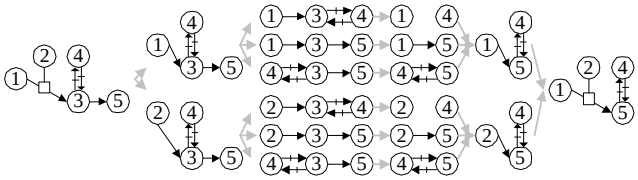


Figure 13. Removing inserted action 3.

After integrating the final actions from figure 14 (in the same way as figure 10) and simplifying the design, we can conclude that the refinement of interaction 'pay' is indeed correct.

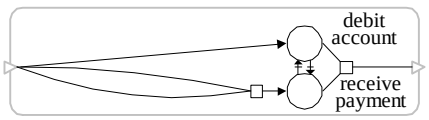


Figure 14. A design with only final actions.

5. Related work

There are various other methods that aim at constructing component assemblies in such a way that these component assemblies conform to the business processes from which they have been derived. We distinguish between two classes of methods.

The first is the class of methods related to the Model Driven Architecture [10]. These methods use algorithms to automatically derive component assemblies from business processes. With these methods, component assemblies automatically conform to the business processes if the algorithm is proven to be correct. The Convergent Architecture [6] uses such a method and provides a tool, named ArcStyler, that supports this method.

The second class of methods, of which our method is one, uses a formal semantics to prove that an assembly of components conforms to the business process of which it is derived. Bowman et al. provide means to check the consistency of enterprise specifications and component assembly specifications in [2]. They use the viewpoints of the Reference Model for Open Distributed Processing (RM-ODP) [7] and their formal semantics as a starting point. However, their work is meant to provide a framework rather than a practical method. The Systemic Enterprise Architecture Methodology (SEAM) [15] also provides a semantics for RM-ODP [9] that can be used to relate business processes to component assemblies. Finally there is the work on modeling the Networked Enterprise by Steen et al. [14]. They use a dialect of ISDL to design business processes and component assemblies. However, to the best of our knowledge, neither SEAM nor the work by Steen et al. defines rigorous refinement techniques.

The language that we use, ISDL, is strongly related to Formal Description Techniques (FDT) a recent survey of FDTs in the area of communication protocols can be found in [1]. The most well-known FDTs that define refinement are LOTOS and Z. In addition to refinement in FDTs, [8] defines refinement for UML models (based on a formal semantics of UML models). [3] and [12] apply refinement to component and business process design, respectively. Our work differs from theirs because we aim at bridging the gap *between* business process and component designs.

6. Concluding remarks

In this paper we have shown a method that helps to develop component-based applications that properly support the business processes in which they will be used.

The method works by introducing an algorithm for conformance verification. This algorithm can be used to verify if an assembly of components conforms to the requirements that are set for it. For the algorithm to work, the requirements have to be expressed in the form of business processes and the assembly of components and the requirements have to be modeled in ISDL. We say that a component assembly properly supports a business process if it conforms to the business process according to the algorithm.

In this paper we have shown the method to work for a simple example. In [11] we proved the correctness of the method with respect to the formal semantics of ISDL.

We are currently working on the practical applicability of the method by developing tool support. A preliminary tool, including a graphical editor and a simulator, for ISDL is available from the homepage of the first author. We are working on tool support for the refinement method described here. Another way in which we are improving the practical applicability of our method is by developing mappings from UML diagrams to ISDL and vice versa, such that the semantics of ISDL and the refinement method described here can be used in UML. We believe that this may help in the construction and verification of model transformations for the Model Driven Architecture.

7. References

[1] Babich, F., Deotto, L. Formal Methods for Specification and Analysis of Communication Protocols. In: *IEEE Communications Surveys and Tutorials* 4(1), 2002.

[2] Bowman, H., Boiten, E., Derrick, J., and Steen, M. Viewpoint Consistency in ODP, a General Interpretation. In: *Proc. of FMOODS*, Paris, France, March '96, 189-204, 1996.

[3] Broy, M. Towards a Mathematical Concept of Component and its Use. In: *Proc. of CUC*, Munich, Germany, July '96, 1996.

[4] D'Souza, D., and Cameron-Wills, A. *Objects, Components, and Frameworks with UML – The Catalysis Approach*. Addison-Wesley, Reading, MA, USA, 1999.

[5] Eertink, H., Janssen, W., Oude Luttighuis, P., Teeuw, W., and Vissers, C. A business process design language. In: *Proc. of the World Congress on Formal Methods in the Development of Computing Systems*, Lecture Notes in Computer Science 1708, Springer, 76-95, 1999.

[6] Hubert, R. *Convergent Architecture*, Wiley, New-York, USA, 2002.

[7] ITU-T / ISO. *Open Distributed Processing Reference Model. Part 1-4*. ITU-T Specification ITU-T 90x, and ISO/IEC Specification ISO/IEC 10746-x, where x = 1..4, 1995.

[8] Jürjens, J. Formal Semantics for Interacting UML Subsystems. In: *Proc. of FMOODS 02*, Enschede, The Netherlands, March '02, 2002.

[9] Naumenko, A. *Triune Continuum Paradigm: a Paradigm for General System Modeling and its Application for UML and RM-ODP*. Ph.D. Thesis, Swiss Federal Institute of Technology, 2002.

[10] OMG. *Model Driven Architecture*. OMG Specification ormsc/02-07-01, 2001.

[11] Quartel, B. *Action Relations – Basic Design Concepts for Behaviour Modelling and Refinement*. Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 1998.

[12] Rumpe, B., and Thurner, V. Refining Business Processes. In: *Proc. of OOPSLA Workshop on Behavioral Semantics*, Vancouver, BC, Canada, October '98, 205-220, 1998.

[13] van Sinderen, M., Ferreira Pires, L., Vissers, C., and Katoen, J.-P. A design model for open distributed processing systems. *Computer Networks and ISDN Systems* 27, 1995.

[14] Steen, M., Lankhorst, M., and van de Wetering, R. Modelling Networked Enterprises. In: *Proc. of EDOC*, Lausanne, Switzerland, September '02, 109-119, 2002.

[15] Wegmann, A. *The Systemic Enterprise Architecture Methodology (SEAM)*. Technical Report EPFL/I&C/200265. Swiss Federal Institute of Technology, 2002.