

Semantic Verification of Behavior Conformance

Remco M. Dijkman, Dick A.C. Quartel, Luís Ferreira Pires, and Marten J. van Sinderen

University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands
{dijkman|quartel|pires|sinderen}@cs.utwente.nl

Abstract. This paper introduces a formal yet practical method to verify whether the behavior design of a distributed application conforms to the behavior design of the enterprise in which the application is embedded. The method allows both enterprise architects and application architects to talk about designs in their own terms, and introduces a common set of terms as the linking pin between enterprise and application designs. The formal semantics of these common terms allows us to verify the conformance between an enterprise and its applications formally and automatically.

1 Introduction

Distributed applications that support the business processes of an enterprise have to be seamlessly integrated with the way in which this enterprise does business. In other words, the applications should implement a part of the enterprise’s behavior, such that the enterprise behavior as a whole is not affected.

In this sense, we can consider the part of the enterprise design that the application implements, as the design of the *desired* behavior of the application. We can then verify if the application design conforms to this desired behavior.

A problem when verifying the conformance of a distributed application design with respect to (the part of) the enterprise design that it implements, is that enterprise architects and application architects both speak their own language. Therefore, they construct designs in their own terminology and design languages. This makes it difficult to relate enterprise and application designs.

A way to solve this problem is to introduce a common design language for enterprise and application design. If we can translate an enterprise design into a design in this common language on one hand, and an application design into a design in this common language on the other hand, then we have a means to relate these designs.

In this paper we introduce such a common language for enterprise and application behavior design. We also explain how enterprise and application behavior designs can be translated to designs in this common language, and how we can verify the conformance of an application design to an enterprise designs via the common language.

Design languages are defined by concepts that architects use to construct a design. A *concept* is an abstraction of system properties that is generalized from system instances. A typical *enterprise concept* is: ‘business process’. An instance of this concept is, for example, ‘the sales process’. A typical *distributed application concept* is: ‘distributed operation’. A design is a composition of concept instances.

To define a common design language, we define a set of *basic concepts*, to which we can map both enterprise and application concepts. An example of a basic concept is: ‘action’, which represents the completion of an activity, where an activity is an organizational unit for performing

a specific function. Since both business processes and operations are specific kinds of activities, we can map their completion to the basic concept action. Therefore, the mapping allows us to reason about the relation between business processes and operations, and we can argue whether or not a composition of operation instances in an application design correctly implements a business process instance in an enterprise design. Figure 1.i shows the relation between the different sets of concepts and the designs constructed with these concepts.

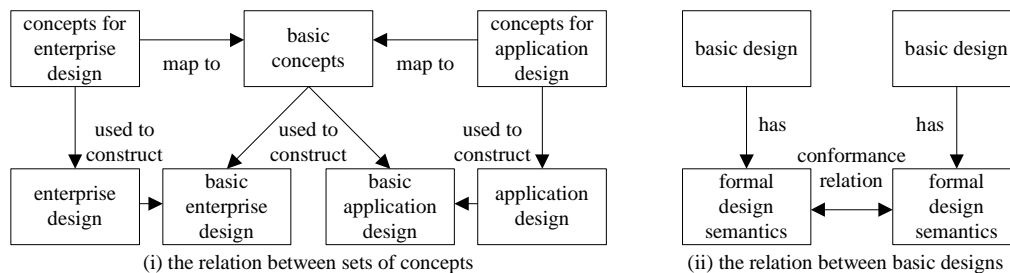


Figure 1: Overview of the Method

We reason about the conformance of an application design to an enterprise design formally, via the formal semantics of the corresponding basic designs. The relation between a basic design and its formal semantics is defined by a mapping from that basic design to a design that is specified in a process algebra. Figure 1.ii shows this.

In this paper, we use the concepts from the Interaction Systems Description Language (Quartel, e.a., 2002), (Quartel, e.a., 1997), and (Sinderen, e.a., 1995) as basic concepts. We chose this set of concepts, because it has proven useful in practice, both for enterprise design and for distributed application design (Eertink, e.a., 1999), (Quartel, e.a., 1999), and (Sinderen, e.a., 1995), and because it has a strong formal semantics (Quartel, 1998).

We identify enterprise design concepts and distributed application design concepts from literature, in particular from standardization efforts in the area of distributed computing (i.e. RM-ODP (ITU-T/ISO, 1995), EDOC (OMG, 2002)). This makes our method specific for enterprise distributed object computing. However, we limit ourselves to design that is independent of specific middleware platforms such as CORBA.

The remainder of this paper is structured as follows. Section 2 explains the basic ISDL concepts and their formal semantics. Sections 3 and 4 explain the concepts for enterprise design and distributed application design respectively. These sections also explain how concepts for enterprise and distributed application design map to basic concepts. Section 5 explains our notion of behavior conformance and shows how formal behavior verification can be performed based on this notion. Section 6 shows a case study with our method, and section 7 discusses related research. Finally, section 8 presents the conclusions of this paper.

2 Behavior Design

This section describes the basic concepts that we use as a basis for design of both enterprise behavior and distributed object behavior. It also discusses their formal semantics.

2.1 Basic Concepts

Our three basic concepts for behavior design are: 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 time moment 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 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 2 shows an example of an action and an interaction. The interaction represents the successful completion of a joint activity of an ATM and its user to get money from the ATM. Two results are obtained in this activity: the amount that is received and the account from which the money is deducted. The completion of the activity occurs at some time moment t, on a location l that is constrained, such that it can only be the ATM interface. The action represents the successful completion of an activity of the ATM to obtain the user's credit status. The figure also shows how we can delimit the behavior of a system (part) by means of a *behavior block*.

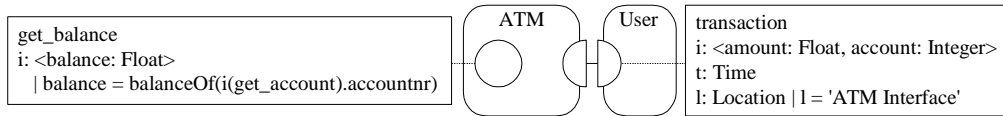


Figure 2: Action and Interaction Example

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: $\textcircled{b} \rightarrow \textcircled{a}$;
- (inter)action *b* must *not* happen before, nor simultaneously with *a*. This is graphically represented as: $\textcircled{b} \nrightarrow \textcircled{a}$;
- (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: $\rightarrow \textcircled{a}$.

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

A *probability attribute* can be added to each sufficient causality condition to represent the chance that the associated (inter)action happens when this condition is satisfied. In this paper we only consider an abstraction of the probability attribute, called the *simple probability attribute* that can have two values: *must* (representing that chance=100%), or *may* (representing that chance<100%). In which case the associated (inter)action either must or may happen if the sufficient causality condition holds. We graphically represent a probability attribute with value

may, by attaching a question mark to the sufficient causality condition. If no question mark is attached to a sufficient causality condition, its probability attribute has the value *must* by default.

Figure 3 shows an example of a set of actions with their corresponding causality conditions. The figure shows that *a* and *c* are always enabled, that *b* may happen after *a*, and that action *d* must happen after *b* or *c* have happened.

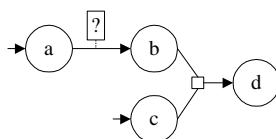


Figure 3: Example Behavior Design

2.2 Formal Semantics

We define the formal semantics $[B]$ of a behavior B by a set that enumerates all possible executions χ of this behavior. Therefore, we also call $[B]$ the *execution-based behavior*. An execution χ is defined as the three-tuple $\langle A, \sim A, < \rangle$, where:

- A is the set of (inter)actions that occur in χ ;
- $\sim A$ is the set of (inter)actions that do not occur in χ ;
- $< \subseteq A \times A$ is the causal relation between (inter)action occurrences, where $(a, b) \in <$ means that the occurrence of a is the actual cause of the occurrence of b ;

For a complete description of the algebra that defines execution-based behavior, and for an algorithm to automatically calculate all possible executions of a behavior from its graphical representation, we refer to (Quartel, 1998). In this paper we rely on an intuitive understanding of the relation between a behavior B and its set of executions $[B]$.

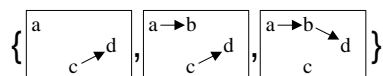


Figure 4: An Example of the Formal Semantics of a Behavioral Design

Figure 4 shows the set of alternate executions that corresponds to the behavior from figure 3. An arrow from an action occurrence a to an action occurrence b represents $(a, b) \in <$. Since b may happen after a has happened, b either happens or not in the set of executions. If it does, it is caused by a . Figure 3 models that d must happen if either b or c has happened. However, when d happens, it happens as a consequence of either b , or c , but not of both. The set of executions shows this by modeling that in one execution b causes d and d is independent of c , while in another execution c causes d and d is independent of b .

3 Concepts for Enterprise Behavior Design

In RM-ODP an enterprise is designed as a community of enterprise objects, formed to achieve a certain goal. An enterprise object may represent any real entity in the enterprise, such as an employee or an application.

The behavior of the enterprise is defined in terms of cooperating roles and business processes. A *role* is an identifier for behavior that may be fulfilled by enterprise objects. A *business process* is also an identifier for behavior, but it differs from a role in the sense that it cannot be directly assigned to enterprise objects. To assign business processes to enterprise objects, we first have to decompose the business process into the roles that will perform it, and then we can assign these roles to the enterprise objects. The behavior of a role is specified in terms of actions, interactions and constraints on the occurrence of actions and interactions. The behavior of a business process that is not yet decomposed into roles, is specified in terms of steps and constraints on the occurrence of these steps. *Steps* are actions that are not (yet) assigned to a role.

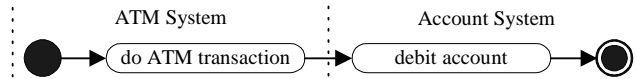


Figure 5: An Example Business Process

Figure 5 shows an example of an enterprise behavior that consists of a single business process modeled as a UML activity diagram. We do not claim that UML activity diagramming is the best way to represent business processes, nor that it adequately represents our set of enterprise concepts. We merely want to use a different notation than ISDL to show that both concepts *and* modeling languages differ for different architects. Figure 5 models that an ATM transaction happens first, after which an account is debited. These two actions are performed by the ATM system and the accounting system role respectively.

The relation between the enterprise behavior concepts and the basic concepts is straightforward. Both roles and processes are mapped to behavior blocks. Enterprise actions and steps are mapped to basic actions. Enterprise interactions are mapped to basic interactions. Constraints are mapped to causality conditions and constraints on attributes. Using this mapping, the basic design from figure 6 corresponds to the business process from figure 5.

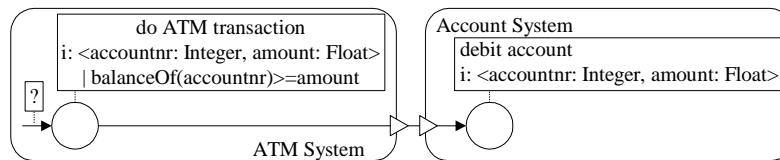


Figure 6: An Example Business Process Described with Basic Concepts

Figure 6 adds some detail to the original business process. It adds the information that is used in the business process, and it adds the information that an ATM transaction may fail, using the question mark.

The triangles pointing into and out of the behavior blocks (called entry and exit points respectively) are needed because we do not allow causality conditions to cross the borders of a behavior block. Therefore, a causality condition may leave one behavior block through an exit point, and then enter another behavior block through an entry point.

4 Concepts for Distributed Application Behavior Design

We design distributed applications in terms of interacting objects. These objects may be physically distributed, and they interact via a middleware platform. The distributed objects interact at interfaces. An *interface* is defined as a subset of the *interactions* of an object with its environment.

In this paper, we only consider *operation interfaces*, which are interfaces on which operations may be invoked. An *operation* (ITU-T/ISO, 1995, part 2 clause 7.1.2) is a specific kind of interaction that is separated into two phases: the call phase, in which information is carried from one object (the calling object) to another (the called object) and the optional return phase, in which information is carried back to the calling object. The called object may either send back a valid return, or a user exception that indicates that the operation failed on the called object's side. Operations may fail independently for the participating objects, meaning that if an operation fails for one object, this object can not be sure whether the operation failed for the other object or not.

If we only define the interfaces of distributed objects, we do not completely describe a distributed application's behavior. To do that, we also have to specify the conditions on which an operation may happen (known as *pre-conditions*), and the conditions that must hold when the operation is done (known as *post-conditions*).

```

interface Bank_ATM_System{
    void do_ATM_transaction(in long accountnr, in float account) raises (BalanceTooLow);
};
interface Account_System{
    float get_balance (in long accountnr);
    void debit(in long accountnr, in float account);
};

```

Figure 7: An Example Distributed Object Interface Design

Figure 7 shows two operation interfaces, specified in CORBA IDL. Again, we do not claim that CORBA IDL is the best way to represent distributed object interfaces, nor that it adequately represents our set of concepts. We only use it to show that application architects typically use a modeling language that varies from the modeling language that business architects use, and that therefore they need a common language to relate their designs to enterprise designs. Figure 7 shows that distributed objects may call ‘do_ATM_transaction’ on the ‘Bank_ATM_System’ interface, thereby requesting the interface to perform an ATM transaction. Objects may call ‘get_balance’, or ‘debit’ on the ‘Account_System’, thereby requesting the balance of an account, or requesting an amount to be deducted from an account. Figure 7 does not specify pre- and post-conditions, because IDL has no way to describe them. Pre- and post-conditions may, however, be specified in comment in the IDL specification. Figure 7 also does not specify calling interfaces.

We map interfaces to behavior blocks, and pre- and post-conditions to causality conditions and constraints on attributes. The mapping from operations to basic concepts is more complex. The composition of basic concepts that correspond to an operation with a return phase is shown in figure 8. The figure shows that after an operation call on the calling side, the middleware passes on the operation call to the called side. This may fail. The called object responds to the operation call either with a return or with a user exception. Either of these is sent back to the calling object by the middleware, and this may again fail. If any passing of information fails, the calling object receives a system exception.

We observe that when we try to write down remote operations as shown in figure 8, the resulting basic designs become difficult to represent and understand. Therefore, we allow the use of shorthands. *Shorthands* are defined by rewrite rules that introduce a single notational element that corresponds to a composition of the basic notational elements. Figure 8 defines the rewrite rule for the distributed operation notational element.

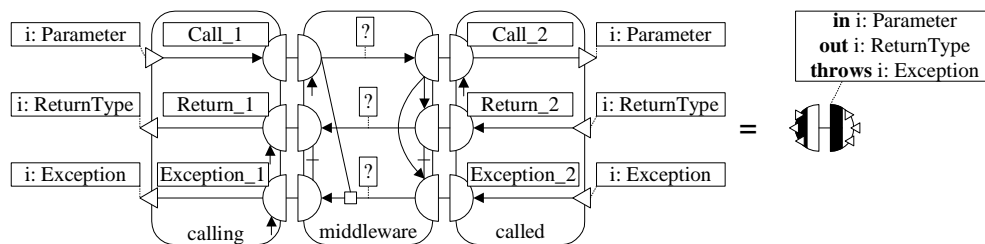


Figure 8: Operation Behavior with Corresponding Shorthand

Using the mapping rules and the shorthand, the basic design from figure 9 corresponds to the distributed application design from figure 7, not including the ‘debit account’ operation. Figure 9 adds some detail to the original design. First, it does show the relations between the operation calls in terms of pre- and post-conditions. Second, it shows the calling part of interfaces. The figure shows that after the bank receives a ‘do_ATM_transaction_call_2’, it performs a request to the account system to obtain the current balance of the account. If the balance is sufficient, then the call is returned. However, if the balance is insufficient, or if a system exception occurs while obtaining the account balance (for example, because the account system is unavailable), then the bank returns an exception. When the exception interaction occurs, it establishes the result:

<e: Exception>. This result may carry different information for each occurrence of the exception interaction, for example, to distinguish between a user exception and a system exception.

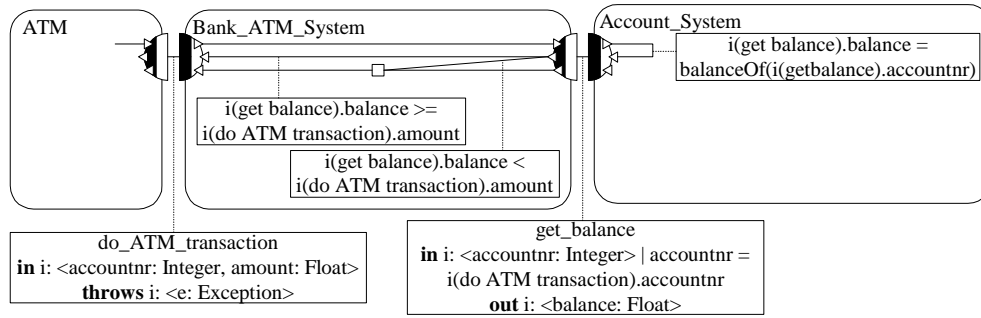


Figure 9: A Basic Distributed Application Independent Design

5 Conformance Verification

Designing applications is a creative process. To create an application design, we typically take a set of user requirements and add information to produce a design, or we take a design and add information to produce a more detailed design. The creative process of adding information to a design is called *refinement*. The design that we refine is called an *abstract design*. The design that is the result of the refinement is called a *concrete design*.

If we want to produce a concrete design that does what the abstract design prescribes, we cannot just add any information to the abstract design, because then, we could add conflicting information, or information that allows more freedom than the abstract design intended. Therefore, we must follow *refinement rules* during the process of refinement. A *refinement step* may consist of multiple applications of these rules. For ISDL, we defined the following refinement rules:

1. action refinement: an action is refined into an activity consisting of multiple actions, or into an interaction (see figure 10.i);
2. causality refinement: a causality condition is refined into multiple causality conditions with actions in between (see figure 10.ii);
3. interaction refinement: an interaction is refined into multiple interactions (see figure 10.iii).

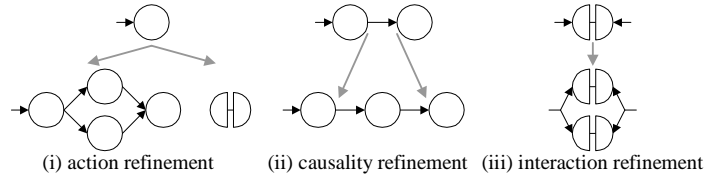


Figure 10: Refinement Rules for ISDL

Our refinement rules give rise to the classification of concrete (inter)actions into inserted (inter)actions and final (inter)actions. *Final (inter)actions* are (inter)actions that correspond to the completion of the abstract (inter)action of which they are a refinement. We distinguish between single final actions, conjunctions of final actions and disjunctions of final actions. We call a final action a single final action, if it alone corresponds to the completion of the abstract action. We call a set of final actions a conjunction of final actions, if the completion of all of these final actions corresponds to the completion of the corresponding abstract action. We call a set of final actions a disjunction of final actions, if the completion of one of these final actions corresponds to the completion of the corresponding abstract action. *Inserted (inter)actions* are concrete (inter)actions that are not final (inter)actions.

A designer may experience the refinement rules as overly restrictive, in particular because refinement is a creative activity. Therefore, we allow for a method to verify afterwards if a

concrete design is correct with respect to an abstract design. This method is based on the observation that, because refinement adds information to an design abstract, we can also abstract from this added information. We then can verify if the design that is the result of abstracting from the inserted information equals the original abstract design. This process of abstracting and checking equality is called *conformance verification*. Figure 11 shows the relation between refinement and conformance verification. We have defined rules to abstract from the information that was added to the abstract design. These abstraction rules are based on the refinement rules, because they represent the inverse of refinement.

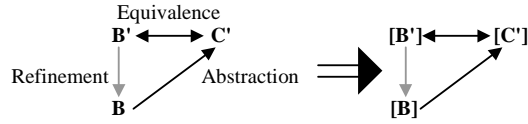


Figure 11: Refinement Verification Approaches

In this paper, we use the formal semantics of designs for conformance verification. We can do this, because if a concrete behavior B conforms to an abstract behavior B' , then the formal semantics $[B]$ of B must conform to the formal semantics $[B']$ of B' . Figure 11 shows the relation between conformance verification of designs and conformance verification of the formal semantics of these designs.

Without proof (for proof and a formal description of the method we refer to (Quartel, 1998)), we define the equivalence of behavioral semantics $[B']$ and $[C']$ as set equivalence ($=$), and we define the abstraction rules to reach an abstract execution $\chi' \in [C']$, from a concrete execution $\chi \in [B]$ as follows.

1. Remove all inserted actions, and use the transitive closure of the causality conditions.
 - 2^a. Replace all single final actions by the corresponding abstract action.
 - 2^b. Replace all conjunctions of final actions:
 - by the occurrence of the corresponding abstract action, if all these final actions occur; or
 - by the non-occurrence of the corresponding abstract action, if one of these final actions does not occur.
 - 2^c. Replace all disjunctions of final actions by the non-occurrence of the corresponding abstract action, if none of these final actions occurs.
- For each action in a disjunction of final actions that occurs in an execution: create a new execution in which this action is replaced by the corresponding abstract action, and in which the other final actions do not occur.
- 2^d. Replace all combinations of disjunctions and conjunctions of final actions, by applying 2^b and 2^c repeatedly.

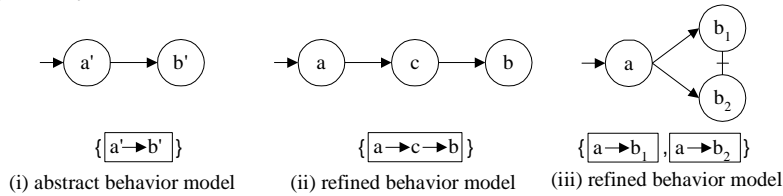


Figure 12: Example of Behavior Refinement Verification

An example of the application of this procedure is shown in figure 12. This figure shows two refinements of an abstract behavior and the corresponding formal semantics. In figure 12.ii, we assume that a is a single final action of a' , b is a single final action of b' and c is an inserted action. According to rule 1, c can be removed which yields $\{a \rightarrow b\}$. Next, we use rule 2a to replace a and b , by a' and b' , which yields $\{a' \rightarrow b'\}$. This is equivalent to the semantics of figure 12.i, and hence figure 12.ii is a correct refinement of figure 12.i. In figure 12.iii, we assume

that a is a single final action of a' , and b_1 and b_2 form a disjunction of final actions of b' . $a \rightarrow b_1$ may then be translated into $a \rightarrow b'$ according to rule 2c. Only one new execution is created, because only one final action occurs. Similarly, $a \rightarrow b_2$ yields $a \rightarrow b'$. So both executions together yield the set $\{a \rightarrow b'\}$, which after applying rule 2a results in $\{a' \rightarrow b'\}$.

6 Case Study

As a case study, consider the business process model from figure 5, and suppose that we want to implement this business process with the distributed application design from figure 7. The corresponding basic designs are shown in figures 6 and 9 respectively.

The execution-based behavior of the business process is simple: either ‘do ATM transaction’ happens, after which the account in question is debited, or nothing happens. This corresponds to: $\{\emptyset, \text{do ATM transaction} \rightarrow \text{debit account}\}$.

We can verify that the application design from figure 9 implements the business process correctly (not including the ‘debit account’ action). Suppose that we performed the refinement by considering ‘do_ATM_transaction_return_1’ as a single final action for ‘do ATM transaction’ in the business process, and inserting all other concrete actions. Whether ‘do ATM transaction’ happens or not then depends on whether ‘do_ATM_transaction_return_1’ happens or not. Figure 13 shows an interaction in which the action can happen.

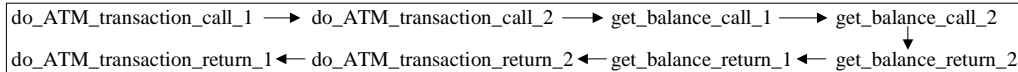


Figure 13: Successful ATM Transaction Execution

‘do_ATM_transaction_return_1’ may also fail for many reasons. For example, because the bank cannot be reached by the ATM machine, or because the account balance is insufficient. Hence the abstract execution corresponding to the implementation is $\{\emptyset, \text{do ATM transaction}\}$, which conforms to the business process behavior, where ‘do ATM Transaction’ either happens or not.

Now suppose that we extend the distributed application design from figure 9 with the debit operation on the account system as shown in figure 14.

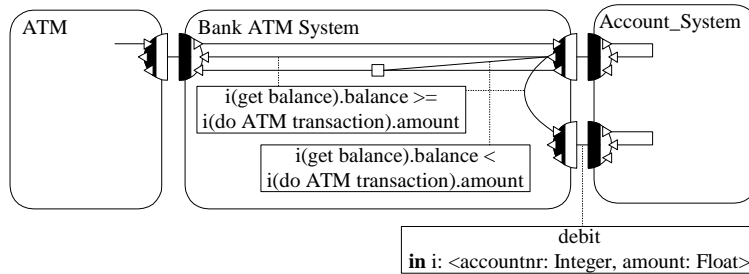


Figure 14: Extending the Implementation of the ATM Transaction Business Process

We can verify that this extension does *not* implement the business process correctly. Suppose that we performed the refinement by considering ‘do_ATM_transaction_return_1’ as a single final action for ‘do ATM transaction’ in the business process, ‘debit_return_1’ as a reference action for ‘debit account’ in the business process, and inserting all other concrete interactions. Then, the abstract execution corresponding to the extended implementation is $\{\emptyset, \text{do ATM transaction}, \text{debit account}, \text{do_ATM_transaction debit_account}\}$. Or in natural language, ‘do ATM transaction’ and ‘debit account’ happen or fail independently. This is wrong because the business

processes states that either ‘debit account’ happens after ‘do ATM transaction’, or neither of the actions happens.

7 Related Work

We believe that our method can help to achieve the ambitious objectives of the Model Driven Architecture (MDA) (OMG, 2001m) approach. The MDA aims at developing applications by creating models from different stakeholder viewpoints and at different levels of abstraction, and relating or mapping them to each other. We have shown that our basic set of concepts can be used to relate behavioral models from different levels of abstraction to each other.

The MDA and some similar approaches (such as the Unified Process (Jakobson, e.a., 1999) and Catalysis (D’Souza and Cameron Wills, 1999)) use the Unified Modeling Language (UML) as a language. However, we argue that the UML is not yet ready for this. The reason is that two models can only be mapped precisely when they have a common, unambiguous semantics (Ciocoiu and Nau, 2000). However, the UML has a semantics that is defined separately for each of its model types, and neither precise (Evans, e.a. 1998), nor unambiguous (Saksena, e.a., 1998) and (Génova, e.a., 2001). Furthermore, two models at different abstraction levels can only be related precisely, if a precise notion of refinement exists. However, UML does not prescribe a refinement relation. Various initiatives exist to solve these problems, such as the precise UML (pUML) that defines a formal semantics for UML, and the action semantics (OMG, 2001a) specification that defines a common semantics for UML behavior models.

The Reference Model for Open Distributed Processing (RM-ODP) (ITU-T/ISO, 1995) provides separate sets of concepts (also called viewpoints) for, among others, enterprise design and computational design. It also defines a set of basic design concepts (ITU-T/ISO, 1995, part 2 clause 8), with which our basic design concepts (action, interaction, activity, and behavior) are aligned. Considerable work was done on defining a formal semantics for RM-ODP (Sinnot, 1997), (Nørbæk and Jørgensen, 1995), (ITU-T/ISO, 1995, part 4), (Johnson and Kilov, 1999) and on defining a formal relation between different RM-ODP viewpoints (Bowman, e.a., 1996), (Bowman, e.a. 1995) and (Taylor, e.a., 2002). However, this work is based on traditional formal techniques for behavior modeling (such as SDL and LOTOS) that were originally meant for protocol specification, and that have problems when describing distributed objects (Pickin, e.a., 1996). We claim that the ISDL concepts are generic enough to describe both protocols and distributed objects.

8 Conclusions and Future Work

This paper presents a method to verify if the behavior of a distributed application design conforms to the behavior of the enterprise in which the application will be embedded. The method is based on the definition of a common set of concepts for enterprise behavior design and application behavior design. In this paper we have shown that this common set of concepts can be used to verify the conformance of designs via their formal semantics.

Although we have only shown our method for the relation between enterprise and application behavior design, the method is generic enough to be applied to other types of behavior design, such as platform dependent distributed application behavior design.

We have not shown in this paper how we can verify the conformance of information, time and location attributes. This topic is discussed in (Quartel, 1998) to a limited extent. However, in addition to the work presented there, we need other methods and notations (such as Z) to verify information, time and location conformance.

To make our method more applicable, we must develop specialized concepts for enterprise behavior design (such as policies), and for distributed application design (such as message streams). In addition to that, we must develop concepts for structural design. The ISDL has means

to describe the structural aspects, but we must develop specialized concepts that are specifically suitable for enterprise and distributed application design.

In this paper we omitted formal proofs, but a formal method to verify the conformance of basic designs exists (Quartel, 1998). If we can combine this formal method with a formal method to map enterprise, and distributed application designs to basic designs, then we can prove formally whether an application design conforms to an enterprise design. Alternatively, we can implement the formal procedure in a tool to automate the proof. Eventually automating the proof will be necessary, because the simple case study presented in this paper already shows that it is not feasible to carry out the conformance verification by hand.

Finally, it will be necessary to perform more case studies, because, while the case study presented here shows that the method *can* be used, it does not demonstrate the practical use, nor the scalability of our method.

References

- Bowman, H., Boiten, E., Derrick, J., and Steen, M. (1996). Viewpoint Consistency in ODP, a General Interpretation, in Najm, E., and Stefani, J.-B. (eds.): *Proc. of the 1st Workshop on Formal Methods for Open Object-Based Distributed Systems*, Chapman and Hall, 189-204.
- Bowman, H., Derrick, J., and Steen, M. (1995). Some Results on Cross Viewpoint Consistency Checking, in Raymond, K., and Armstrong, L. (eds.): *Proc. of the IFIP TC6 International Conference on Open Distributed Processing*, Chapman and Hall, 399-412.
- Ciocoiu, M., and Nau, D. (2000). Ontology-Based Semantics. In Cohn, A., Giunchiglia, F., and Selman, B. (eds.): *Proc. of Knowledge Representation and Reasoning*, Morgan Kaufmann, San Francisco, 539-546.
- D'Souza, D., and Cameron-Wills, A. (1999). *Objects, Components, and Frameworks with UML – The Catalysis Approach*. Addison-Wesley, Reading, MA.
- Evans, A., Bruel, J.-M., France, R., Lano, K., and Rumpe, B. (1998). Making UML Precise. In Andrade, L., Moreira, A., Deshpande, A., and Kent, S. (eds.): *Proc. of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*
- Eertink, H., Janssen, W., Oude Luttighuis, P., Teeuw, W., and Vissers, C. (1999). A business process design language. In Wing, J., Woodcock, J., and Davies J. (eds.): *Proc. of the World Congress on Formal Methods in the Development of Computing Systems*, Lecture Notes in Computer Science 1708, Springer, 76-95.
- Génova, G., Llorens, J., and Martínez, P. (2001). Semantics of the Minimum Multiplicity in Ternary Associations in UML. In Gogolla, M., and Kobryn, C. (eds.): *Proc. of the 4th International Conference on UML – Modeling Languages, Concepts, and Tools*, Springer, Berlin, 329-341.
- ITU-T / ISO (1995). *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.
- ITU-T / ISO (1999). *Information Technology - Open Distributed Processing Reference Model – Enterprise Language*. ITU-T Specification ITU-T 911, and ISO/IEC Specification ISO/IEC 16414.
- Jacobson, I., Booch, G., and Rumbaugh, J. (1999). *The Unified Software Development Process*. Addison-Wesley, Reading, MA.
- Johnson, D., and Kilov, H (1999). An approach to a Z toolkit for the Reference Model of Open Distributed Processing. *Computer Standards and Interfaces* 21, 393-402.
- Nørbæk, B., and Jørgensen, M. (1995). Modifying and Using SDL to Specify ODP-Based Telecommunications Services. In Bræk, R., and Sarma, A. (eds.): *SDL'95 with MSC in CASE*, Elsevier Science, Amsterdam, 211-220.
- OMG (2001). *Action Semantics for the UML*. OMG Specification ad/2001-08-04.
- OMG (2001). *Model Driven Architecture*. OMG Specification ormsc/02-07-01.

- OMG (2002). *UML Profile for Enterprise Distributed Object Computing Specification*. OMG Specification ptc/02-02-05, and ad/01-08-20.
- Pickin, S., Sánchez, C., Yelmo, J., Gil, J., and Rodríguez, E. (1996). Introducing Formal Notations in the Development of Object-Based Distributed Applications. In Najm, E., and Stefani, J.-B. (eds.): *Proc. of the 1st Workshop on Formal Methods for Open Object-Based Distributed Systems*, Chapman and Hall.
- Quartel, D., Ferreira Pires, L., and van Sinderen, M. (2002). On architectural support for behavior refinement in distributed systems design. Accepted for publication in: *Journal of Integrated Design and Process Science*.
- Quartel, D., van Sinderen, M., and Ferreira Pires, L. (1999). A model-based approach to service creation. In: *Proc. of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, IEEE Press, 102-110.
- Quartel, B. (1998). *Action Relations – Basic Design Concepts for Behaviour Modelling and Refinement*. Ph.D. Thesis, University of Twente, Enschede, The Netherlands.
- Quartel, D., Ferreira Pires, L., van Sinderen, M., Franken, H., and Vissers, C. (1997). On the role of basic design concepts in behaviour structuring. *Computer Networks and ISDN Systems* 29, 413-436.
- Saksena, M., France, R., and Larrondo-Petrie, M. (1998). A Characterization of Aggregation. In: *Proc. of the 5th International Conference on Object Oriented Information Systems*.
- van Sinderen, M., Ferreira Pires, L., Vissers, C., and Katoen, J.-P. (1995). A design model for open distributed processing systems. *Computer Networks and ISDN Systems* 27, 1263-1285.
- Sinnott, R. (1997). *An Architecture Based Approach to Specifying Distributed Systems in LOTOS and Z*. Ph.D. Thesis, University of Stirling, United Kingdom.
- Taylor, C., Boiten, E., and Derrick, J. (2002). Interpreting ODP Viewpoint Specification: Observations from a Case Study, in Jakobs, B., and Rensink, A. (eds.): *Proc. of the 5th Workshop on Formal Methods for Open Object-Based Distributed Systems*, Kluwer, 61-76.