# Customer Validation of Formal Contracts

Rogardt Heldal and Kristofer Johannisson

Chalmers University of Technology
Gothenburg, Sweden
[heldal|krijo]@cs.chalmers.se

**Abstract.** This paper shows how to write formal OCL contracts for system operations in such way that a translation to natural language (a subset of English), understandable by a customer, can be obtained automatically. To achieve natural language text understandable by a customer we use the vocabulary of the problem domain when writing formal contracts for system operations. The benefits of our approach are that we increase the precision of the model by using formal specifications, and that a customer is able to validate (by viewing the natural language rendering) if a contract actually describes the behavior desired from the system. Without validation of this kind there is generally no guarantee that the formal specification states the correct properties.

## 1 Introduction

Large programs need specifications. These specifications might be used in different contexts: for software developers to support the implementation of the software, for testers to understand the required behavior of the software, for customers to validate the correctness of the system, and for users of the software to understand the behavior of the system.

It is important that the specifications are of high quality to avoid problems such as ambiguity and under-specification. To that end, several formal languages have been developed to write more precise specifications [1, 4, 5]. The nature of these languages forces one to be more precise than when using natural language to specify behavior of programs. The problem is, however, that not everyone involved in the software process — for example the customers — can be expected to understand these formal languages. So, the customer cannot, at least easily, validate whether a formal specification states the correct behavior or not. There is little point in being precise if the specification states wrong properties. This is the problem we deal with in this paper.

We consider formal contracts similar to those in Eiffel [11] by attaching OCL (Object Constraint Language)[12] constraints to *system operations* in UML [12] class diagrams. The constraints specify the signature of an operation together with its pre- and post-conditions. We previously developed a tool[3] which translates OCL constrains to natural language text (a subset of English). But removing the overhead of OCL does not necessary make the natural language text

understandable for customer and users. The text might contain too many design and implementation details and is therefore more suited for designers and implementers.

This paper is about controlling the vocabulary used in OCL pre- and post-conditions for system operations, in such a way that natural language text produced by our tool can be read by customers and users having domain knowledge, but not necessarily computer science knowledge. Not only does our tool make translation understandable for customers, users, and domain experts, but it also permits the tool to be used earlier in a software development process than previously [3]. With respect to our previous work in [9, 3], the contribution of this paper is not to improve the tool itself, but rather a new use of the tool.

In our previous work [3] the focus was on the quality of the translation for any type of OCL contracts. In this work we restrict ourself to contracts for system operations, which represent the external interface of the system. The reason for only considering these operations is that customers and users are mostly interested in these operations, and not in the internals of the system. It is crucial that the contracts for system operations capture the behavior customers want. It is impossible to build a system correctly if one does not know its expected behavior. It is well known that requirement deficiencies are the prime source of project failures [7].

We introduce the notion of *abstract contracts*, which abstract away from implementation details by using *problem domain models* instead of class diagrams for system operations contracts. Both problem domain models and class diagrams can be represented by UML class diagrams, but they differ in an important way: the vocabulary of domain models should be fully understood by the customer but it is not a requirement for class diagrams used in the design phase. Domain models restrict OCL constraints to a vocabulary, which should be completely understood by the customer. Translating these formal contracts using our tool does not only remove OCL, but also guarantees that the natural language text obtained contains vocabulary of the domain model — the vocabulary of the customers, users, and domain experts. So the only requirement for reading these specifications are domain knowledge which permits validation of the abstract formal contracts without having to read OCL — the customer only needs to read the natural language text produced by our tool.

There are further benefits of our work going beyond the validation of formal specifications. The natural text produced can be used as a documentation of the system. Since the text is generated from formal specifications, even the natural language text will be in some sense formal, but of course readable for customers. So, the benefit is more precise natural language specifications avoiding ambiguity and under-specification. Furthermore, if the formal specifications change, one can just produce new natural language text, avoiding the problem of synchronizing formal and informal specifications.

One important lesson to learn from our work is that if the translation being for most parts compositional, as in our case, from formal to natural language the choice of vocabulary of the formal specification becomes crucial in controlling

the vocabulary of the natural text produced. Customer- and user-understandable natural text does not come for free. In our case the trick was to use the domain model.

The key contributions: (1) We create a formal, machine-transformable model early in the software engineering process (2) The model can be validated by customers, since the OCL contracts can be translated into understandable natural language. (3) By automatically translating OCL to natural language and using OCL as a single source, we avoid the problem of synchronizing formal specifications with natural language text describing customer requirements. (4) The natural text produced can be used as part of the documentations of systems.

*Paper Overview.* Some background on the translation tool and domain models is given in Sect. 2 and 3. We then discuss system operations, contracts and the vocabulary of the domain model in Sect. 4, 5 and 6. Finally, Sect. 7 describes related work and we conclude in Sect. 8.

## 2 From OCL To Natural Language

We have developed a tool for linking specifications in OCL to natural language specifications. It has been previously described in [9], where we give basic motivation and design principles, and in [3], where we conduct a case study. It is based on the Grammatical Framework (GF) [13], and is being integrated into the KeY system [2]. The basic idea of the tool is to define an abstract representation (an abstract syntax) of "specifications" and to relate this representation to both OCL and English using a GF grammar. The GF system then allows us to translate between OCL and English using the abstract syntax as an interlingua. We can therefore always keep OCL and English in sync.

Given a UML model, the tool provides two basic functionalities: (1) Automatic translation of OCL specifications into English. (2) A multilingual, syntax-directed editor which allows editing of OCL and English specifications in parallel. The input to the translator is an OCL specification, along with a description of the UML model (a domain model or class diagram) in question. The output is English text, formatted in HTML or LaTeX. In the editor, OCL and English are kept in sync since the editing takes place on the level of abstract syntax.

In previous work [3] we did not consider the style of formal abstract contracts. These contracts contained design and implementation details, and we refer to them as *concrete contracts*, for an example see Fig. 1. This figure shows parts of the OCL specification for the operation *check* of the class *OwnerPIN*, a class in the Java Card API. Java Card [17] is a subset of Java, tailored to smart cards and similar devices, which comes with its own API. The operation *check* checks whether a given PIN matches the PIN on a smart card, keeps track of the number of times you have entered an incorrect PIN, and so on.

Translating concrete contracts to natural language text removes the overhead of having to understand OCL, but they still contain design and implementation details, see Fig. 2. This figure shows the natural language translation provided

```
context OwnerPIN::check(pin: Sequence(Integer),
         offset: Integer, length: Integer): Boolean
post: (self.tryCounter > 0 and not (pin <> null and offset >= 0
            and length >= 0 and offset+length <= pin->size()
            and Util.arrayCompare(self.pin, 0, pin, offset, length) = 0)
      ) implies (not self.isValidated()
            and self.tryCounter = tryCounter@pre-1 and
            (( not excThrown(java::lang::Exception) and result = false)
               or excThrown(java::lang::NullPointerException) or
               excThrown(java::lang::ArrayIndexOutOfBoundsException)))
```

**Fig. 1.** Design Level OCL Contract

by our system. The translation will make sense only to a reader who already has an understanding of the Java Card API classes. He or she must be familiar with the use of byte arrays and their representation in OCL, as well as with various Java Card exceptions.

For the operation **check ( pin : Sequence(Integer) , offset : Integer , length : Integer ) : Boolean** of the class **javacard::framework::OwnerPIN** , the following post-condition should hold :

- if the try counter is greater than 0 and at least one of the following conditions is not true
  - *pin* is not equal to null
  - *offset* and *length* are at least 0
  - *offset* plus *length* is at most the size of *pin*
  - the query `arrayCompare ( the pin , 0 , pin , offset , length )` on `Util` is equal to 0

  then this implies that the following conditions are true
  - this own er PIN is not validated
  - the try counter is decremented by 1
  - at least one of the following conditions is true
    - `Exception` is not thrown and the result is equal to false
    - `NullPointerException` is thrown
    - `ArrayIndexOutOfBoundsException` is thrown

**Fig. 2.** Design Level Natural Language Contract

In summary, concrete contracts, in formal or natural language, are not generally understandable to customers, since customers cannot be expected to be familiar with design and implementation matters.

If we compare the natural language text in Fig. 2 to the original OCL contract in Fig. 1, we can note that they both share roughly the same structure

and vocabulary. This is a consequence of our translation being (for most parts) compositional, using the same abstract syntax for both OCL and natural language. If an OCL contract uses the vocabulary of a UML class diagram (design phase), which is not normally understandable to a customer, the natural language translation will not be understandable either. This is fine since concrete contracts are meant to be read and understood by designers and implementers but not customers and users.

The goal of this work is to obtain specifications understandable by customers, users, and domain experts not trained in computer science. We show how domain models can be used for this purpose, but first we will have closer look at what we mean by a domain model.

## 3 Domain Models

In this section we provide background on domain models used in our work. The problem domain models we use are as in the book of Larman [10]. A problem domain model is a visualization of concepts in a real-life domain of interest without behavior, not software classes such as Java, C++, or C# classes.

UML does not offer separate notation for domain models, but a restricted form of UML class diagrams can be used. For example, the operation compartment of UML classes are not needed. On the other hand, the benefit is that this permits OCL constraints to be written over the domain models. As a running example we will consider an ATM system. The concepts involved in such a system would include *Card*, *Customer*, *Account*, and *Bank*, as shown in Fig. 3. In general, a banking application would require more concepts and attributes, but for our purposes this domain model is sufficient. To make our example simpler, we assume that a customer can never have more than one account.

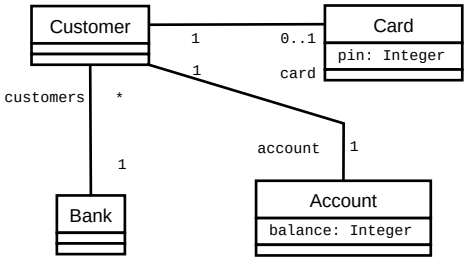


**Fig. 3.** Domain model of an ATM machine

Concepts can have attributes. For example, a *Card* has a PIN code, and an *Account* has a balance. Furthermore, concepts can stand in relationship to each other; in our example a *Card* is associated with *Customer*. With the help of multiplicity annotations, one can describe constraints on how many instances of

one concept are related to another. For example, one can express the constraint that each card can be linked to only one customer.

There are no hard and fast rules of how to choose the correct abstraction level for domain models. But there is one rule which should never be broken: whatever abstraction level one chooses, the domain model should always be understandable for customers, users, and domain experts. Of course, the more concrete and detailed domain model the more concrete contracts can be written. Yet, a domain model should never be so concrete and detailed that the customer does not understand it anymore.

Before considering how domain models permit to create formal contracts rather early in the development process we will have a closer look at system operations.

## 4 System Operations

System operations are the operations that deal with the events coming from outside the system. For information about how to obtain system operations from use cases we refer to [15]. From a customer's point of view system operations are the important operations: they define the functionality of the system. To exemplify we consider three operations for our ATM system: *identify*, *authenticate*, and *withdraw*, as seen in Fig. 4. We collect the system operations in a class *ATMController*, further explained in Sect 5.

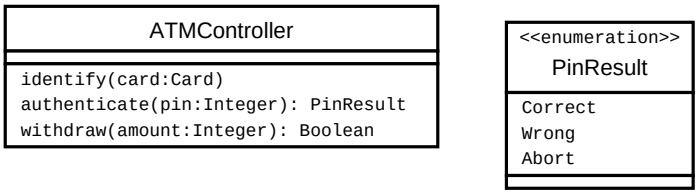


**Fig. 4.** System Operations for an ATM

The identity of a customer is given by presenting a card, using the operation *identify*, the operation *authenticate* ensures that the card is used by the authorized person, and *withdraw* is used to retrieve money from the ATM machine. We introduce an enumeration type to indicate the result of *authenticate*: either the pin is accepted (*Correct*), or it is wrong but the user is given another chance to enter the correct code (*Wrong*), or the wrong code has been given too many times (*Abort*). The result of *withdraw* is a boolean, indicating whether there was enough money in the account or not. In next section we will consider how to obtain contracts for these system operations.

## 5 Abstract Formal Contracts

To write OCL contracts for system operations requires a context for the operations. Our approach is to include all system operations in one UML class and associate this class to appropriate concepts of the problem domain — creating a hybrid between a class representing the system and concepts. This permits us to write OCL pre- and post-conditions for the system operations. It might be necessary to add attributes to the system class to model the state of the system not captured by the domain model. In our running ATM example, we put the system operations in a class *ATMController* and attach it to the domain model as shown in Fig. 5. We have also added two attributes *numberOfTries* and *pinAccepted* to keep track of the state.

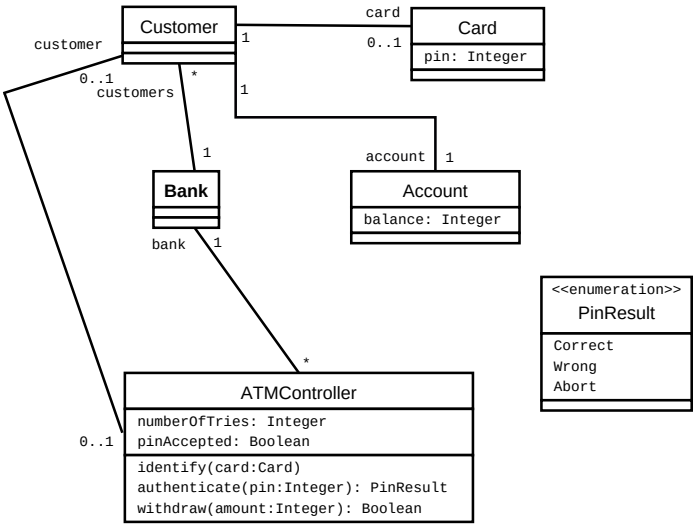


**Fig. 5.** The System Class Attached to the Domain Model

The UML class diagram in Fig. 5 can be viewed as the first approximation of the system to be built, but it is important to point out that it is not the final software design. In an object oriented design one will expect that some or all of the concepts of Fig. 5 to become design classes with operations and possibly more attributes. Furthermore, new classes will most often be added due to for example design patterns. The purpose of the class diagram in Fig. 5 is to permit writing formal contracts for system operations using the vocabulary of the customer.

Fig. 6 shows abstract OCL contracts for our example ATM system operations using the class diagram of Fig. 5. Let us consider the operation *identify*. This operation should initialize the state of *ATMController*: the association from

```
context ATMController::identify(card:Card)
post: customer = card.customer
      and numberOfTries = 0
      and not pinAccepted


context ATMController::authenticate(userPin:Integer) : PinResult
pre: not pinAccepted
post: numberOfTries = numberOfTries@pre + 1
      and if userPin = customer.card.pin and numberOfTries <= 3
          then pinAccepted and result = PinResult::Correct
          else not pinAccepted
               and if numberOfTries <= 3
                   then result = PinResult::Wrong
                   else result = PinResult::Abort
                   endif
          endif

context ATMController::withdraw(amount:Real) : Boolean
pre: pinAccepted
post: if (amount <= customer.account.balance)
      then customer.account.balance =
                customer.account.balance@pre - amount
           and result = true
      else customer.account.balance =
                customer.account.balance@pre
           and result = false
      endif
```

**Fig. 6.** OCL Abstract Contracts

*ATMController* to *Customer* is instantiated with the customer of the card, the number of tries is 0, and a correct pin code has not yet been entered.

Obtaining a customer from the association of the card is not enough. In addition, a guarantee that one is dealing with the right customer is desirable. According to the domain model the concept *Card* is related to a customer. So, the operation *authenticate* compares the PIN given as argument with the PIN stored in the card associated with the customer. Depending on whether the PIN argument matches the PIN on the card, and on the number times an incorrect PIN has been entered, the state of the system class *ATMController* is set appropriately.

Finally, there is *withdraw* which gives the customer money if there is enough money in the account. The return value indicates whether the withdrawal was successful or not.

Both problem domain model and system operations are found early in the development process, so abstract contracts like the one in Fig. 6 can be created

at an early stage of the development process. Even though the contracts in Fig. 6 are not readable for people not trained in formal methods, we will see that the natural language counterpart will indeed be readable.

Using the domain model in the contracts also provides a validation of the domain model, which is important since the domain model is the foundation of the system to be built. One might discover shortcomings — e.g. a missing attribute or concept — when creating the contract. In this sense, writing the contract over the domain model may improve the domain model itself.

## 6 The Vocabulary of the Domain Model

Fig. 7 shows what our translator produces from the OCL specification in Fig. 6. Natural language contracts should be understandable to customers not only because it has been translated from OCL to natural language, but also because it uses a vocabulary of the problem domain model.

If we compare the natural language text in Fig. 7 to the original OCL contract in Fig. 6, we can note that they both share roughly the same structure and vocabulary. Again, this is a consequence of our translation being (for most parts) compositional, using the same abstract syntax for both OCL and natural language. In comparison to the natural language text produced in Fig. 2 the text in Fig. 7 do not contain any design and implementation details.

Improving the style of the natural language text is still ongoing research. For example, we can see from Fig. 5 that fragments of the formal abstract contract can be found in the natural language text, such as $identify(card : Card)$. We have not found a way of translating this fragment better and more clear than just keeping the method signature.

It is important to point out that formal abstract contracts such as in Fig. 6 probably have to be produced by formal methods experts, but customers and users only need to consider the natural language counterpart.

In this work we are not directly considering the problem of formalization: constructing formal specifications from informal ones. However, once an OCL contract has been developed, our tool can be used to generate a natural language contract, which can then be validated against the informal specification. This can be used to improve the original informal specification as well as the formal one.

## 7 Related Work

The approach taken in [16, 15] is similar to ours in the sense that they also provide OCL contracts for system operations. An important difference to our work is that they do not consider customer validation of the OCL contracts, which reduces the benefit of using formal contracts at an early stage in the process. They also consider how to find the system operations based on descriptions of use cases, which we do not discuss. Our work is more focused on the domain model and the translation to natural language.

For the operation **identify (card: Card)** of the class **ATMController** ,
the following post-condition should hold:

- the following conditions are true
  - the customer is equal to the customer of *card*
  - the number of tries is equal to 0
  - the property pinAccepted does not hold

---

For the operation **authenticate (userPin: Integer): PinResult** of the class **ATMController** ,
given the following pre-condition:

- the property pinAccepted does not hold

then the following post-condition should hold:

- the number of tries is incremented by 1 and if *userPin* is equal to the pin of the card of the customer and the number of tries is at most 3 then:
  - the property pinAccepted holds and the result is equal to `Correct`

  otherwise:
  - the property pinAccepted does not hold and if the number of tries is at most 3 then:
    - the result is equal to `Wrong`

    otherwise:
    - the result is equal to `Abort`

---

For the operation **withdraw (amount: Real): Boolean** of the class **ATMController** ,
given the following pre-condition:

- the property pinAccepted holds

then the following post-condition should hold:

- if *amount* is at most the balance of the account of the customer then:
  - the balance of the account of the customer is decremented by *amount* and the result is equal to true

  otherwise:
  - the balance of the account of the customer does not change and the result is equal to false

**Fig. 7.** Contracts in Natural Language

The UP process [10] also makes use of contracts for system operations, which are based on the domain model. However, the contracts are informal. We believe that rather than having another informal specification, system operation contracts are a good place to become formal. Our approach could be incorporated into the UP process by replacing informal system operation contracts with formal ones. In that way one can introduce formal specification into the UP process at an early stage in the development process.

Previously, we have also been working on relating formal and informal specifications [6]. In that paper we gave a method for relating post-conditions of use cases to OCL contracts. In that paper, we did not consider different styles of writing contracts or translation to natural language text. Furthermore, we required formal method experts to use our method of relating formal and informal specifications.

The idea of producing natural language from a formal representation to enable validation by people not trained in formal languages is familiar from conceptual modelling and requirements engineering. For instance, the paper [8] presents a system for generating natural language explanations from conceptual models, and also gives an overview of related work. The basic difference to our approach is that we translate textual OCL contracts for system operations — not the domain model itself — into natural language, while [8] generates explanations from "... process-oriented and static ER-like languages...", e.g. data flow diagrams. As explained in Sect. 6, the translations we provide have the same basic structure as the OCL specifications, and are understandable to a customer because of the use of the vocabulary of the domain model. In contrast, the process of generating explanations from conceptual models in [8] involves information extraction from the model, and various strategies for presenting this information in natural language. There is also work on going in the other direction: from informal, natural language text to conceptual models, e.g. [14].

## 8 Conclusion and Future Work

We have presented a way of attaching system operations to a domain model which permits the creation of formal abstract contracts, which in turn can be translated to natural language understandable to a customer. The purpose is to provide a precise model at an early stage which can be validated by customers. Formal specifications require more precision than informal ones, such as informal contracts and use cases. So, the developer has to make decisions about the behavior of the system. Our approach permits the customer to validate that the decisions made in the formal specification are really what the customer wants. This is crucial to provide a good starting point for the development process.

Since we have an automatic translation from OCL to natural language, we can always keep formal and natural language contracts synchronized. This is in general considered a very hard task. We have also identified the importance of using domain models when writing contracts, in particular with respect to the readability of natural language translation of the contracts.

As future work, we want to look at how to relate our work to Model Driven Architecture (MDA) [12]. Maybe our abstract contract could be a good starting point for MDA transformations. At some point models need to be related to informal specifications.

In addition, we plan to test how real customers react to our generated texts. The vocabulary should be known to the customers since the generated text contains the vocabulary of the domain model. However, the style of the generated text might be confusing.

We also plan to further investigate the quality of the natural language text produced. For example, given a system to be specified one can have groups writing informal contracts and other groups writing formal abstract contracts. Thereafter our tool would translate the formal abstract contracts to natural language contracts. This permits comparison between informal text produced by hand and the text produced by our tool.

Another interesting experiment would be to start from an informal specification, and then create an abstract formal specification. Then our tool would be used to translate the abstract formal specification into natural language text. Thereafter, make a comparison between the two natural language specifications.

## References


1. J. R. Abrial. *B-Book*. Cambridge University Press, 1996.
2. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.
3. David Burke and Kristofer Johannisson. Translating formal software specifications to natural language—a grammar-based approach. In P. Blache, E. Stabler, J. Busquets, and R. Moot, editors, *Logical Aspects of Computational Linguistics*, number 3492 in LNAI. Springer, 2005.
4. International Organisation for Standardization. Information technology—programming languages, their environments and system software interfaces, Vienna Development Method, specication language, part 1: Base language, 1996. ISO/IEC 13817-1.
5. International Organisation for Standardization. Information technology—Z formal specification notation—syntax, type system and semantics, 2000. ISO/IEC 13568:2002.
6. Martin Giese and Rogardt Heldal. From informal to formal specifications in UML. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc. of UML2004, Lisbon*, volume 3273 of *LNCS*, pages 197–211. Springer, 2004.
7. Robert L. Glass. *Software Runaways. Lessons Learned from Massive Software Project Failures*. Prentice Hall, 1998.
8. Jon Atle Gulla. A general explanation component for conceptual modeling in CASE environments. *ACM Transactions on Informal Systems*, 14(3), 1996.
9. Reiner Hähnle, Kristofer Johannisson, and Aarne Ranta. An authoring tool for informal and formal requirements specifications. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, number 2306 in LNCS, 2002.

10. C. Larman. *Applying UML and Patterns, second edition*. Addison-Wesley, 2002.
11. B. Meyer. *Object-Oriented,Software Construction*. Prentice Hall PTR, 1997.
12. OMG. *Unified Modeling Language Specification version 1.5 (2.0)*, 2005. `http://www.omg.org/technology/documents/formal/uml.htm`.
13. Aarne Ranta. Grammatical Framework: A type-theoretical grammar formalism. *The Journal of Functional Programming*, 14(2):145–189, 2004.
14. Colette Rolland and C. Proix. A natural language approach for requirements engineering. In *Advanced Information Systems Engineering, 4th International Conference CAiSE '92*, volume 593 of *LNCS*. Springer, 1992.
15. Shane Sendall and Alfred Strohmeier. From use cases to system operation specifications. In Stuart Kent and Andy Evans, editors, *UML'2000 — The Unified Modeling Language: Advancing the Standard, Third International Conference*, volume 1939 of *LNCS*. Springer, 2000.
16. Alfred Strohmeier, Thomas Baar, and Shane Sendall. Applying fondue to specify a drink vending machine. In *Proceedings of OCL 2.0 Workshop at UML'03*, volume 102 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.
17. Sun Microsystems. Java Card homepage, 2005. `http://java.sun.com/products/javacard/`.