A Formal Development and Validation Methodology for System Design

Giovanna Di Marzo Serugendo
University of Geneva, Computing Centre, CH-1211 Geneva, Switzerland
LGL-DI, Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland

ABSTRACT

This paper presents first a formal development methodology that enables the validation of formal specifications as well as program codes wrt client's requirements. The methodology is based on the two languages framework: it advocates the joint use of a model-oriented specification and a logical language. Second, the paper describes development guidelines for system design within the proposed methodology. Guidelines, specific to each kind of system, can be defined. They enable the specifier to add progressively the complexity into the system design. Two development processes are proposed: the first one leads to a traditional client/server design; the second one enables to integrate fault-tolerance in the design. Both development processes have been applied on an example and produced a Java program.

Keywords: Structuring Complex Concurrent Systems, Formal Development, Stepwise Refinement, Development Guidelines, System Design, Java.

1. INTRODUCTION

The engineering of distributed systems should be based on a development methodology with a design phase relying on well-established design principles and formal enough for a serious verification phase to be defined.

Formal methods traditionally use a single formal specifications language for expressing both the requirement specifications, and the system specifications. When the chosen formal specifications language is property-oriented (e.g., a logical language), the specification task is more difficult, but the verification task is reduced to showing logical implications. When the chosen formal specifications language is model-oriented, specifications are more easily and powerfully expressed, but the verification task is difficult and usually follows an informal way (e.g., simulation).

In order to bring a solution to the problem of the choice between a model-oriented and a property-oriented formal specifications language, some model-oriented specifications languages have acquired a property-oriented specifications language. This is known as the *two languages framework* described, among others, by Pnueli in [14]: a logical language is used for expressing requirements, and a model-oriented language is used for describing models or implementations. In addition, the logical language is also used for translating the system specification into logical properties, and the verification task is then realized in the logical framework. Among others, the VDM^{++} [11] language and the temporal Petri nets of [8] use this approach.

This paper presents a *formal development methodology* that follows the two languages framework. It addresses the three classical phases of the development process of distributed applications: the analysis phase, the design phase, and the implementation phase. The design phase relies on the stepwise refinement of formal model-oriented specifications. The correctness of the refinement steps and the validation of formal specifications and programs wrt client's requirements is verified on the basis of logical formulae.

The proposed methodology is general enough and can be applied to any model-oriented formal specifications language. This paper describes the particular application that has been realised for a special kind of synchronized Petri nets, called Concurrent Object-Oriented Petri Nets (CO-OPN/2) [2]. CO-OPN/2 is an object-oriented formal specifications language that allows concurrent systems to be described in terms of structured Petri nets for the behavior part, and algebraic specifications for the data structures used to define values managed by the Petri nets. Temporal logic formulae used for the refinement steps are expressed by means of the Hennessy-Milner logic (HML).

This paper describes as well *development guidelines for system design* within the proposed methodology. Two development processes are proposed: the first one leads to a traditional client/server design; the second one enables to integrate fault-tolerance in the design. Distributivity and implementation constraints are progressively added during the development process. Each step is formally proved using logical formulae. Thus, every specification, as well as the obtained programs, validate the initial requirements.

The structure of this paper is as follows. Section 2 describes the formal development methodology, and presents its application to synchronized Petri nets. Section 3 illustrates the proposed formal development methodology and the use of development guidelines during the design phase. Section 4 presents related work.

2. DEVELOPMENT METHODOLOGY

The methodology proposed in this paper follows the two languages framework. It addresses the three classical phases of the development process of distributed applications: the *analysis* phase, the *design* phase, and the *implementation* phase.

After the analysis phase, informal requirements are determined. The design phase consists of the stepwise refinement of model-oriented specifications. The methodology advocates for this

phase the joint use of model-oriented specifications and logical formulae. The behavior of a system is specified by means of model-oriented specifications. Such specifications provide a model of the system, and implicitly define a set of properties corresponding to the behavior defined by the specification. During a refinement step it is not always necessary to preserve the whole behavior proposed by the specification. Therefore, essential properties expected by the system are explicitly expressed by means of a set of logical formulae, called *contract*. A contract does *not* reflect the whole behavior of the system, it reflects only the behavior part that must be preserved during all subsequent refinement steps. A refinement is then defined as the replacement of an abstract specification by a more concrete one, which respects the contract of the abstract specification, and takes into account implementation constraints.

Finally, the implementation phase is treated in a similar way as the design phase. At the end of the design phase, a concrete model-oriented specification is reached, it is implemented, and the obtained program is considered to be a correct implementation if it preserves the contract of the most concrete specification.

Figure 1 shows the three phases. On the basis of the informal requirements, an abstract specification Spec_0 is devised. Its contract $\mathrm{Contract}_0$ formally expresses the requirements. During the design phase, several refinement steps are performed, leading to a concrete specification Spec_n and its contract $\mathrm{Contract}_n$. The implementation phase then provides the program $\mathrm{Program}$ and its contract $\mathrm{Contract}$. A refinement step is correct if the concrete contracts contain the abstract contracts.

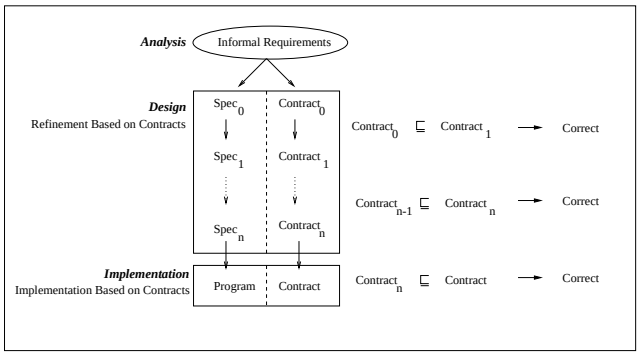

Figure 1. Development Methodology

The particularity of our methodology wrt traditional ones using the two languages framework is that: first, it goes a step further, since the contracts explicitly point out the essential properties to be verified. The specifier can then freely refine the formal specifications, without being obliged to keep all the behavior. Second, the methodology covers the whole development process since it addresses also the implementation phase. Third, the use of contracts enables to formally prove the validation of formal specifications and programs wrt client's requirements.

General Theory

In order to formally support the methodology defined above, a general theory of refinement and implementation based on contracts has been defined [4]. The theory applies to any model-oriented formal specifications language and any logical language that enables to express formulae on these specifications.

Refinement Based on Contracts: As we intend to make explicit the use of properties in order to constrain the refinement, we require every specification to be linked with a set of properties. This set of properties is called a *contract*. The pair formed by a specification and a contract is called a *contractual specification.* The contract is actually a set of formulae, expressed on the specification, that is satisfied by all models of the specification.

The basic idea of refinement consists in replacing a high-level (abstract) contractual specification by a lower-level (more concrete) contractual specification whose models preserve the contract guaranteed by the higher-level specification.

We do not constrain syntactically the lower-level contractual specifications wrt the higher-level ones, i.e., syntactical changes are allowed. A *refine relation* associates one or more elements of the high-level contractual specification to elements of the lower-level contractual specification. The refine relation explains the syntactical evolution of the high-level specification towards the lower-level specification. The use of a refine relation, allowing syntactical changes, implies the translation of the high-level contract into a set of formulae expressed on the lower-level specification. The translation is performed by the means of a *formula refinement*, i.e., a function, univocally defined on the basis of the refine relation, which maps every high-level property of the contract into a lower-level formula. The formula refinement explains the semantical evolution of the high-level specification to the low-level specification, e.g., when a high-level element is related to several lower-level elements, the formula refinement has to explain how the lower-level elements replace the single higher-level element in a formula.

The refinement is then defined as the replacement of a high-level contractual specification by a lower-level contractual specification whose contract *contains* the *translated* contract of the higher-level contractual specification. In this way, every model of the lower-level specification satisfies the translated contract of the higher-level specification, since it satisfies the contract of the lower-level specification. The lower-level contract may contain the "same" formulae (via the translation) as the higher-level one. It can also contain *more* formulae; the additional formulae describe additional constraints that have to be verified by all further refinement steps.

Implementation Based on Contracts: A refinement step consists in replacing a high-level specification by a lower-level specification, both specifications being expressed within the *same* language. The implementation step replaces a specification by a program, expressed in a programming language, which is usually *different* from the specifications language. The implementation links the world of specifications to the world of programs. Thus, the implementation shares a lot of similarities with the refinement.

The basic idea of implementation consists in replacing a contractual specification by a *contractual program* whose models preserve the contract of the contractual specification. A contractual program is defined like a contractual specification, it is a pair made of a program and a contract, i.e., a set of properties that the program guarantees.

We do not constrain syntactically a low-level specification wrt a high-level specification. Due to the change of language, the gap between the program and the specification is bigger than that between two specifications. Thus, we will neither constrain syntactically the program wrt the contractual specification. An *implement relation* associates elements of the contractual specification to elements of the contractual program. Formulae of the specifications are translated to formulae expressed on the programs, by the means of a function called *formula implementation.*

The implementation is then defined as the replacement of a contractual specification by a contractual program whose contract *contains* the *translated* contract of the contractual specification.

Summary: Figure 2 shows a refinement process followed by an implementation phase, and depicts the proofs necessary to ensure that the whole process is correct.

The refinement process starts with the pair $CSpec_0 =< Spec_0, \Phi_0 >$ as the most abstract contractual specification. A first refinement leads to the pair $CSpec_1 =< Spec_1, \Phi_1 >$. Relation λ_0 is the refine relation on $CSpec_0$ and $CSpec_1$, and Λ_0 is the formula refinement on contract Φ_0.

The refinement process continues and reaches the pair $CSpec_n =< Spec_n, \Phi_n >$. Finally, the implementation phase provides the contractual program $CProg =< Prog, \Psi >$. Relation λ^I is the implement relation on $CSpec_n$ and $CProg$, and Λ^I is the formula implementation on contract Φ_n.

Horizontal proofs ensure that every pair $CSpec_i =< Spec_i, \Phi_i > (0 \leq i \leq n)$ obtained during the refinement process is actually a contractual specification, and that $CProg =< Prog, \Psi >$ is actually a contractual program. Therefore, it is necessary to show that a contract is satisfied by all models of the specification:

$$Mod_{Spec_i} \models \Phi_i \ (0 \leq i \leq n), \text{ and } Mod_{Prog} \models \Psi.$$

Vertical proofs assert the correctness of the refinement steps, by requesting the inclusion of translated contracts:

$$\Lambda_i(\Phi_i) \subseteq \Phi_{i+1} \ (0 \leq i \leq n-1).$$

Finally, *implementation proof* ensures that the contractual program $CProg =< Prog, \Psi >$ correctly implements the contractual specification $CSpec_n =< Spec_n, \Phi_n >$, and hence every contractual specification $CSpec_i \ (0 \leq i \leq n)$. It requests, similarly to vertical proof, the inclusion of contracts:

$$\Lambda^I(\Phi_n) \subseteq \Psi.$$

Application of the Theory to CO-OPN/2

We present here the application of the general theory described above to a model-oriented specifications language based on Petri nets, called CO-OPN/2. Formulae are expressed using the Hennessy-Milner branching time temporal logic.

CO-OPN/2: CO-OPN/2 is an object-oriented formal specifications language [2] that integrates Petri nets used to describe concurrent behaviors and algebraic specifications [16] of structured data evolving in Petri nets.

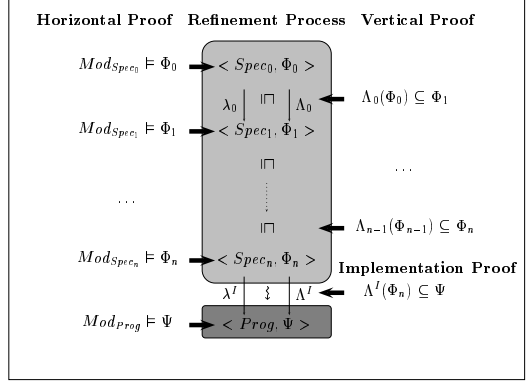

Figure 2. Theory of Refinement and Implementation

An object is considered to be an independent entity composed of an internal state which provides some services to the exterior. The only way to interact with an object is to invoke one of its services; the internal state is thus protected against uncontrolled accesses. CO-OPN/2 defines an object as an encapsulated algebraic net in which places compose the internal state and transitions model the concurrent events of the object. A place consists of a multiset of algebraic values. Transitions are divided into two groups: parameterized transitions, also known as methods, and internal transitions. The former correspond to the services provided to the outside, while the latter describe the internal behavior of an object. Unlike methods, internal transitions are invisible to the exterior world and spontaneous (they are fired as soon as their pre-conditions are satisfied). An object method can only be fired if no further internal transition can. A class describes all the components of a set of objects and is considered to be an object template. Thus, all the objects of one class have the same structure. Objects can be dynamically created. Objects are also called class instances. Each class instance has an identity, which is also called an object identifier, that can be used as a reference.

When an object requires a service, it asks to be synchronised with the method (parameterized transition) of the object provider. The synchronisation policy is expressed by means of a synchronisation expression, which can involve many partners joined by three synchronisation operators (one for simultaneity (//), one for sequence (..), and one for alternative or non-determinism (+)). For instance, an object may simultaneously request two different services of two different partners, followed by a service request to a third object.

Hennessy-Milner Logic: HML formulae are expressed on CO-OPN/2 specifications. An HML formula is a sequence (or a conjunction ($\wedge$), or an alternative (+)) of observable events. Such an event is either the firing of a single method of a CO-OPN/2 object, or the parallel firing of several methods. An HML formula is satisfied by the model of a CO-OPN/2 specification if the sequence of events defined by the formula correspond to a possible sequence of events of the model of the specification.

HML formulae on object-oriented programs are defined similarly to HML formulae on CO-OPN/2 specifications.

Theory: In order to apply the general theory of refinement and implementation based on contracts to the CO-OPN/2 language, and the Hennessy-Milner logic, it is necessary to define: the refine relation on CO-OPN/2 specifications, and the corresponding formula refinement; as well as the implement relation on CO-OPN/2 specifications and object-oriented programs, and the corresponding formula implementation. These relations and functions are such that:

- *Refine relation*: it acts like a *function* (every abstract element is related to at most one concrete element); it is *injective* (two abstract elements cannot be related with the same concrete element); *partial* (an abstract element may have no corresponding concrete element); but *total on elements that appear in the contract* (every abstract element that appear in the abstract contract must be related to a concrete element). The refine relation imposes the preservation of the part of the abstract specification that is necessary for establishing the abstract contract (as well as its object-oriented structure). It allows renamings, as well as the removal or addition of elements of the specification.
- *Formula Refinement*: it is a simple rewriting of the formulae of the abstract contract, based on the renaming given by the refine relation;
- *Implement relation*: it is, similarly to the refine relation, a simple renaming of the elements of a CO-OPN/2 specification into elements of an object-oriented program;
- *Formula Implementation*: it is a rewriting of HML formulae on specifications into HML formulae on programs.

Example: We consider the following simple example: computing the sum of the integers present in a multiset. The computing of the sum follows the Gamma paradigm [1]: a chemical reaction removes two values from a multiset, computes their sum and inserts the result into the multiset.

Class DSGammaSystem

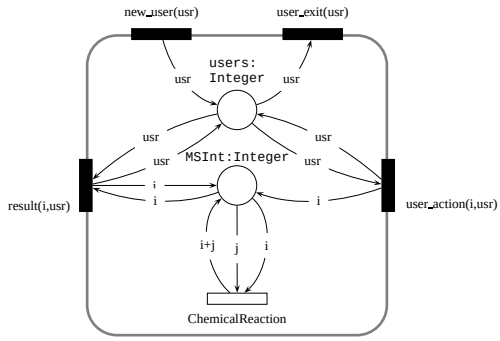

Figure 3. DSGamma System

CO-OPN/2 specifications are graphically noted in the following manner: a CO-OPN/2 class is depicted as a rectangle with a circle for each place inside, a white rectangle for each internal transition, and, on its sides, a black rectangle for each method. Labelled arrows between places and internal transitions or between places and methods give the flow relations (what is consumed and what is added to a place when an internal transition or a method is fired).

Figure 3 shows the `DSGammaSystem` class. Four system operations are provided: they are specified by four CO-OPN/2 methods. Method `new_user(usr)` enables a new user `usr` to be added to the system at any moment. Method `user_action(i,usr)` enables `usr` to add integer `i` into the system. Method `user_exit(usr)` enables `usr` to leave the system. Method `result(i,usr)` enables `usr` to see the computed result `i`.

The internal transition `ChemicalReaction` actually performs the Gamma chemical reactions: it removes two integers from place `MSInt`, sums them up, and inserts the result into the place.

We present a contract ϕ made of only two HML formulae: ϕ_1 and ϕ_2.

Formula ϕ_1 states that after the creation of a DSGammaSystem DSG, it is possible to insert a first user usr_1, and a second user usr_2, after that usr_1 inserts integer i in the system, usr_2 inserts integer j, and usr_2 can see the correct result $i + j$.

Formula ϕ_2 is similar but usr_1 leaves the system before usr_2 sees the result. The exit of usr_1 does not affect the computing of the sum.

$$
\begin{aligned}
\phi_1 = \; & <DSG.\mathrm{create}><DSG.\mathrm{new\_user}(usr_1)> \\
& <DSG.\mathrm{new\_user}(usr_2)><DSG.\mathrm{user\_action}(i, usr_1)> \\
& <DSG.\mathrm{user\_action}(j, usr_2)><DSG.\mathrm{result}(i + j, usr_2)> \\
\phi_2 = \; & <DSG.\mathrm{create}><DSG.\mathrm{new\_user}(usr_1)> \\
& <DSG.\mathrm{new\_user}(usr_2)><DSG.\mathrm{user\_action}(i, usr_1)> \\
& <DSG.\mathrm{user\_action}(j, usr_2)><DSG.\mathrm{user\_exit}(usr_1)> \\
& <DSG.\mathrm{result}(i + j, usr_2)> \; .
\end{aligned}
$$

Formulae ϕ_1 and ϕ_2 actually form a contract, since they correspond to a possible run of the specification given by Figure 3 (transition `ChemicalReaction` occurs in a non observable way as soon as two integers are present in place `MSInt`).

3. SYSTEM DESIGN

This section describes two development guidelines: the first one enables to reach a client/server design; the second one a dependable design. Both guidelines are illustrated with an example, based on the computing of the sum of integers presented in the previous section. Starting with the same abstract contractual specification, two different Java programs are developed.

Development Guidelines

The theory of refinement and implementation based on contracts provides the basis to formally prove that a refinement step and the implementation phase are correct. However, the theory cannot help the specifier in establishing a contract, and in choosing a more concrete specification. Therefore, we suggest the use of *development guidelines*, i.e., a sequence of refinement steps that a specifier should follow when developing an application. Development guidelines depend on the kind of application being developed, they help the specifier in addressing and solving complex problems progressively during the refinement process.

Client/Server Applications

A client/server application is a distributed application with at least two entities: a server processing requests, and one or more clients requiring services from the server. The clients and the server are physically distributed over a network and a communication layer (TCP/IP sockets, UDP sockets, RPC, etc.) is necessary.

Guidelines: The development steps identified in the case of client/server applications are the following:

1. a set of informal application's requirements including validation objectives is defined;
2. an initial contractual specification based on the informal requirements provides an abstract view of the application, where the problem is *neither distributed nor decentralized*. The contract reflects the functionality of the application;
3. a first refinement step provides a decentralized view of the application, where the *data is distributed* but not yet the behavior. The contract takes into account the fact that data is distributed, as well as added features depending on this distribution;
4. a second refinement step provides the application with a *client/server architecture* where both the data and the behavior are distributed. The contract must integrate the client/server architecture, and any protocol or algorithm that the specifier applied when this architecture is used;
5. a third refinement step introduces the *communication layer*. In addition, it introduces features of the programming language. This step leads to a view *close to the code*, i.e., it takes into account the semantics of the programming language. The contract contains the whole set of properties satisfied by the specification;
6. the last step produces the *program* directly from the last refinement step.

Example: The application to develop is based on the computing of the sums of integers as described by the DS-Gamma system of Figure 3. The refinement process according to the guidelines above [5] are the following:

1. *informal requirements*: the application to develop allows several users to insert integers into a multiset that is distributed across the Web. The application computes the sum of integers present in the system. Every user can see the result;
2. *initial contractual specification*: the specification part is given by Figure 3. The system is made of a single global multiset. Every user inserts integers into this multiset. The computing of the sum is realized on the integers of the multiset, by repeatedly or simultaneously removing two integers from the multiset, and inserting their sum into the multiset.

   The contract contains every formulae ensuring that the computing of the sum is correct;
3. *first refinement*: the system is made of several multisets (one for each user). Every user inserts integers into his corresponding multiset. The computing of the sums randomly removes integers from one or two multisets and inserts their sum into a multiset. The integers present in the local multiset of a user who wants to leave the system must be properly dispatched to the other local multisets.

   The contract extends the initial contract with every formulae necessary to ensure the correctness of the computing when users leave the system;
4. *second refinement*: the system is made of several applets, and a server. Each user is mapped to a dedicated applet: the applet is responsible for maintaining a local multiset of integers; and for allowing the user to insert integers into this multiset. The computing of the sum is realised in two steps: first, every applet sends the integers maintained in its local multiset to the server; second, every applet collects pairs of integers from the server, makes up their sum, and inserts the result into its local multiset. The applet has to correctly send its local multiset of integers to the server, once the user wants to leave the system. The applets have to avoid a deadlock situation that would occur when the number of integers present in the whole system is less than the number of applets; a timeout is introduced in order to avoid this situation.

   The contract takes into account the client/server architecture (applets, server) as well as the deadlock situation;
5. *third refinement*: the system is made of the applets, the server, of TCP/IP based sockets between the applets and the server, and all necessary threads handling the sockets. A more sophisticated quitting protocol than that used in the second refinement is needed, in order to avoid that integers are lost because they remain in sockets when a user leaves the system. The timeout used to avoid the deadlock situation is maintained. In addition, this step integrates the Java virtual machine semantics.

   The contract takes into account the TCP/IP sockets as well as the Java virtual machine semantics;
6. *implementation*: the Java program is simply derived from the third refinement.

Dependable Applications

By dependable applications we mean applications that are able to cope with some software or hardware faults. In the particular case of dependable distributed applications, a design phase has been identified where the dependability and distribution constraints are built according to the *Coordinated Atomic action* concept.

Coordinated Atomic (CA) action model: It provides structuring primitives and support for error recovery in designing systems composed of several concurrent interacting entities [17]. The model distinguishes between cooperative concurrency, which is expressed using *conversations* [15], and competitive concurrency, which is expressed using *transactions* [10]. Conversations are used to control cooperative concurrency and to implement coordinated and disciplined error recovery while transactions are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency.

Each CA action has a set of roles that are activated by action participants (external activities such as threads, processes), which

cooperate within the CA action scope. Logically, a CA action starts when all roles have been activated and finishes when all of them have reached the CA action end.

External (transactional) objects can be used concurrently by several CA actions in such a way that information cannot be smuggled among these CA actions and that any sequence of operations on these objects bracketed by the start and completion of a CA action has the ACID (atomicity, consistency, isolation and durability) properties [10] with respect to other sequences. To the outside world, the execution of a CA action looks like an atomic transaction.

The state of a CA action is represented by a set of local objects; each CA action deals with these objects to guarantee their state restoration if error recovery is to be provided. Local objects are the main means for participants to interact and to coordinate their execution (external objects can be used as well).

The CA action mechanism also provides a basic framework for exception handling, which can support a variety of fault tolerance mechanisms aimed at tolerating both hardware and software faults [3, 12].

The use of CA action design makes it easier to prove formally that the system has certain properties, since each CA action guarantees a set of properties. These can be used to construct the proof of global system properties.

Guidelines: The design phase of dependable distributed applications using the CA action concept is as follows [6]:

1. a set of informal application requirements is stated. It includes validation objectives as well as fault tolerance constraints;
2. an initial contractual specification is built. It gives a model of the application, which is abstract enough to be *as independent as possible of implementation constraints*, i.e., distributivity and fault-tolerance are not taken into account. The contract represents the functionality of the application;
3. a refinement of the initial specification is realised. It provides the *CA action design* of the application where distributivity and fault-tolerance are integrated. The contract takes into account the CA action design as well as the faults that the application is able to recover;
4. the next step provides a specification *close to the code* that takes into account the semantics of the programming language. The contract is made of all formulae satisfied by the specification;
5. finally, the *implementation* is derived.

Example: This example is similar to the example used for the client/server application. In addition, the system has to cope with errors that can occur during the computing of the sums.

1. *informal requirements*: the application to develop allows several users to insert integers into a distributed multiset. The addition operation can fail, the storage and removal of integers into/from the multiset can fail;
2. *initial contractual specification*: it is given by the same contractual specification as the client/server example. The system is made of a single global multiset, and every user inserts integers directly into this multiset. The computing of the sum is realized on the integers of the multiset. This contractual specification assumes that no errors occur;
3. *first refinement*: the system is composed of a set of participants (one per user) that maintain a local multiset; a set of CA actions; and a CA action scheduler that handles CA actions. A CA action is made of three roles: two participants (producers) that furnishes each one integer from their local multiset, and a participant (consumer) that collects these two integers, sumps them up, and stores the result into its local multiset. Local multisets of the participants are external objects, and can be accessed only within CA actions. Therefore, consistency and integrity are guaranteed by the CA action support. In the case of failure of the addition operation, the consumer keeps both integers by inserting them into its local multiset, while the producers complete the CA action as if no error had occur. The CA action scheduler is responsible for receiving information from all participants about any new number they have in their local multiset. It starts a new CA action whenever there are at least two integers in the local multisets. There can be as many CA actions active concurrently as there are pairs of integers in all local multisets at a given time.

   The contract integrates the failure of the addition operation;
4. *second refinement*: the system follows the same CA action design as the first refinement. In addition, it integrates the Java Remote Method Invocation (RMI) in order to distribute objects. The CA action scheduler is a remote object, participants are remote applets. CA actions are objects whose roles are methods.

   The contract takes into account the RMI communication as well as the Java virtual machine semantics;
5. *implementation*: the Java program is derived directly from the second refinement.

4. RELATED WORKS

Definitions of refinement of formal specifications are usually not related to an implementation phase, and not incorporated into a development methodology. This section describes first, some definitions of refinement of formal specifications that are close to the definition given in this paper; second, a definition of implementation.

Specifications

Definitions of refinement are all based on the preservation of (possibly translated) properties (either implicitly, or explicitly by the means of additional logical formulae). We present first the definitions of the refinement of formal specifications (VDM^{++}, and Timed Petri nets) that are similar to our approach. Indeed, they use refine relations and additional formulae to verify the correctness of the refinement steps. However these definitions differ from our, since they are based on a whole set of properties to be preserved instead of a customized contract. Second, we give the traditional definitions of refinement of Petri nets.

VDM : It is an object-oriented specifications language due to Lano [11]. The definition of refinement is based on the following idea: "If D is a refinement of C, it must not be possible for a user of the common interface to be able to devise an experiment which would allow him to deduce whether he had an instance of C or of D." This implies the following: D must not remove functionality or behavior from C, and D can add new methods only if the behavior of the new methods can be described as a combination of the behavior of methods of C.

In order to formally prove a refinement step, for each class C a logical RTL (Real Time Logic) language $\mathcal{L}_C$ is defined, and a theory Γ_C expressing the semantics of C in this language is given. Similarly for D, a theory Γ_D is given. The refinement is defined on the basis of these theories. D refines C via a retrieve function R, from the attributes of D to those of C, and a renaming ϕ of the methods of C, if:

$$\forall \psi \in \mathcal{L}_C, \Gamma_C \vdash \psi \;\Rightarrow\; \Gamma_D \vdash \phi(\psi[R(v)/u]).$$

This formula means that the translation in D of every formula ψ that is true in the theory of C leads to a formula that is still true in the theory of D. The translation of a formula in C consists in replacing each attribute of C appearing in the formula by the corresponding expression of D (built with attributes of D) given by the retrieve function R, and by renaming the methods using ϕ.

Timed Petri nets: We present now an approach concerning the refinement of timed Petri nets based on the use of a temporal logic. TRIO is a linear, first-order typed temporal logic due to Ghezzi *et al.* [9]. A TRIO axiomatisation, due to Felder *et al.* [8], has been given to a kind of timed Petri nets where each transition is associated with a firing time interval describing its earliest and latest firing time after enabling. A transition consumes exactly one token from each place in its preset, and produces exactly one token into each place in its postset. At a given time a transition may fire several times.

The TRIO axiomatisation of these timed Petri nets is based on two predicates: (1) $nFire(v, n)$ means that, at the current time, transition v fires n times, and (2) $tokenF(s, i, p, v, j, d)$ means that, at the current time, the ith firing of transition s produces a token that enters place p, this token is consumed after d time units by the jth firing of transition v. Given a net N, a set of axioms $Ax(N)$ is built, that takes into account the net and its initial marking. From $Ax(N)$ a theory is derived, noted $\mathcal{N}$. On the basis of the two above predicates and arithmetic operators, formulae can be expressed over the net. If a formula ϕ can be derived from the theory of N, $\vdash_{\mathcal{N}} \phi$, then *every* execution of the net satisfies the property ϕ.

The implementation relation, of Felder *et al.* [7], of a net S by a net I, is based on the *preservation* of *observable* properties. A net I implements a net S if the observable properties of S are also observable properties of I after translating them into I. The only observable events in a net are transition firings. Therefore, an observable property ϕ, of a net S, is a formula constructed on the basis of the firing predicate $nFire(v, n)$ only, and must be derived from $\mathcal{S}$ (the theory of S): $\vdash_{\mathcal{S}} \phi$.

During a refinement step, it is possible to refine a transition by several transitions (not just one). An event function, $\lambda : T_I \rightarrow T_S$, maps transitions of I to transitions of S. The event function may be partial (a transition of I has no corresponding transition in S), has to be surjective (every transition in S must have at least one corresponding transition in I, so that every observable property of S can be translated into an observable property of I). The event function may be non-injective: a transition in S may be associated to several transitions in I.

Given an event function λ, a property function, $\Lambda : \mathcal{S} \rightarrow \mathcal{I}$ is univocally derived. It translates properties of the theory $\mathcal{S}$, of S, to properties of the theory $\mathcal{I}$, of I. The translation is based on the translation of the firing predicate:

$$\begin{aligned}\Lambda(nFire(v, n)) = \exists n_1 \ldots \exists n_s (n_1 + \ldots + n_s = \\ n \wedge nFire(v_1, n_1) \wedge \ldots \\ \wedge\, nFire(v_s, n_s))\end{aligned}$$

where $\{v_1, \ldots, v_s\}$ is the set of all transitions of I mapped to v ($\lambda(v_i) = v, 1 \leq i \leq s$). The predicate that asserts that transition v fires n times is translated into a predicate that says that the sum of firings of the transitions of I mapped to v is also n.

A net I implements a net S through λ iff for each observable formula ϕ of S:

$$\vdash_{\mathcal{S}} \phi \;\Rightarrow \vdash_{\mathcal{I}} \Lambda(\phi).$$

This means that every observable formula of S is translated into an observable formula of I.

Petri nets: The techniques for refining unstructured Petri nets are based on the replacement of a transition or a place by a subnet. These techniques differ in the way the subnet is embedded inside the initial net. Moreover, we can distinguish two families of refinement techniques: (1) techniques ensuring that the initial net and its refinement have the same properties (they are equivalent in some sense); (2) techniques ensuring that, given an equivalence relation, two equivalent nets are refined to two equivalent nets. Techniques of the first family (a net refines to an equivalent net) are used when both the original net and its refinement have the same behavior. Techniques of the second family (two equivalent nets refine to two equivalent nets) are used when the refinement introduces new elements, such that the original nets and their respective refinements have different behaviors.

Programs

The verification that a program is correct wrt system specifications is a problem similar to the one of verifying that system specifications are correct wrt the requirement specifications. Thus, the use of a logical language in addition to a programming language should help the verification task. In the last decades, only few attempts have been undertaken to consider the idea of integrating assertions into programs. More recently, Meyer [13] has promoted this idea, and even goes a step further. Indeed, he advocates that, in order to face the problem of correctness, every program operation (instruction or routine body) should be systematically accompanied by a pre- and a post-condition.

5. CONCLUSION

This paper presents a methodology for developing distributed programs based on the stepwise refinement of formal specifications. The methodology advocates the use of specific logical properties for guiding and verifying refinement steps, and enables progressive system design and formal validation wrt client's requirements. The paper presents as well development guidelines that help the specifier to face problems progressively, since additional complexity is introduced at each step.

Further research will consider:

- development guidelines that address other kinds of applications, e.g., systems developed using mobile agents;
- development guidelines as refinement patterns. Development guidelines should be considered as an essential means for developing correct systems;
- automated verification, and construction of contracts;
- the improvement of the Hennessy-Milner logic with some temporal operators, or even the use of another logic in the framework of CO-OPN/2. Indeed, the Hennessy-Milner logic is a very simple logic that enables to easily build tools for verification. However, it lacks expressivity, since any invariant needs an infinite set of formulae to be described.

6. ACKNOWLEDGMENTS

The author wants to thank Nicolas Guelfi for his help in assessing the formal methodology, as well as Alexander Romanovsky and Avelino Zorzo for having provided the CA actions design of the example.

7. REFERENCES

[1] J.-P. Banâtre and D. Le Métayer. Gamma and the chemical reaction model. In IC Press, editor, *Proceedings of the Coordination'95 workshop*, 1995.

[2] O. Biberstein, D. Buchs, and N. Guelfi. CO-OPN/2: A concurrent object-oriented formalism. In *Proc. Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. Chapman and Hall, 1997.

[3] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.

[4] G. Di Marzo Serugendo. *Stepwise Refinement of Formal Specifications Based on Logical Formulae: from CO-OPN/2 Specifications to Java Programs*. PhD thesis, Swiss Federal Institute of Technology in Lausanne, 1999.

[5] G. Di Marzo Serugendo and N. Guelfi. Formal Development of Java Based Web Parallel Applications. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS'31)*. IEEE Computer Society, 1998.

[6] G. Di Marzo Serugendo, N. Guelfi, A. Romanovsky, and A. F. Zorzo. Formal development and validation of Java dependable distributed systems. In *Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'99)*, 1999. Submitted.

[7] M. Felder, A. Gargantini, and A. Morzenti. A theory of implementation and refinement in timed Petri nets. *Theoretical Computer Science*, 202(1–2):127–16, 1998.

[8] M. Felder, D. Mandrioli, and A. Morzenti. Proving properties of real-time systems through logical specifications and Petri net models. *IEEE Transactions on Software Engineering*, 20(2):127–141, February 1994.

[9] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO, a logic language for executable specifications of real time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.

[10] J. Gray and A. Reuter. *Transaction processing: concepts and techniques.* Morgan Kaufmann Publishers, San Mateo, CA, USA, 2nd edition, 1993.

[11] K. Lano. *Formal Object-Oriented Development.* Springer-Verlag, London, 1995.

[12] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 1990.

[13] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[14] A. Pnueli. System specification and refinement in temporal logic. In R. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *LNCS*, pages 1–38. Springer-Verlag, 1992.

[15] B. Randell. Systems structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[16] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 13, pages 675–788. North-Holland, Amsterdam, 1990.

[17] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *Proceedings of the 25th Int. Symp. on Fault-Tolerant Computing*, pages 450–457. IEEE CS Press, 1995.