Exploiting Various Levels of Semantics in CO-OPN for the SANDS Environment Tools

Didier Buchs, Jarle Hulaas and Pascal Racloz

Swiss Federal Institute of Technology
Software Engineering Laboratory
1015 Lausanne Ecublens
Switzerland

email: {Didier.Buchs, Jarle.Hulaas, Pascal.Racloz} @ di.epfl.ch
phone: +41 (21) 693.27.37 - fax: +41 (21) 693.50.79

Abstract This paper describes the techniques and tools developed to support the construction of CO-OPN specifications (Concurrent Object Oriented Petri Nets) [7] and the kind of semantics that are needed for each part of the environment. CO-OPN is a specification language which permits an abstract description of aspects of concurrent operations and data-structures of computer programs. The concurrent and operational part of the formalism is described by means of Petri nets, while the data-structures are described by means of algebraic abstract data types. In the CO-OPN formalism, these associations are structured with the notion of objects. CO-OPN specifications can be developed, analyzed and simulated in a dedicated environment called SANDS (Structured Algebraic Nets Development System [6]). Various semantics of CO-OPN are exploited: an abstract operational semantics is used for the logic programming simulator as well as for the test set generation tool, while a more concrete operational semantics based on transactions is necessary for prototyping on distributed systems. All these semantics are valid and partially complete on a subpart of the CO-OPN language with respect to its denotational semantics.

Keywords: Object-orientation, distributed systems, formal methods, Petri nets, algebraic specifications, evolutive prototyping.

1 Introduction

At present, one of the major difficulties faced in software engineering is the production of correct distributed products. The CO-OPN project [7] (Concurrent Object-Oriented Petri Nets) was undertaken in order to propose a formal specification language for the design of large distributed systems. The latter was created by means of a modular specification language in combination with Petri Nets. The underlying formalism, called hierarchical algebraic nets, used as a semantic description of the CO-OPN language, can be seen as a modular extension of algebraic nets. In CO-OPN, hierarchical algebraic net modules are called *objects*. Therefore, the aim of the CO-OPN specification language is to assure the production of software of good quality by means of a complete development method based on the life cycle, including the specification, the validation, verification and implementation phases (cf. figure 1).

This paper describes the environment named SANDS, which was developed to assist in the construction of CO-OPN specifications, but emphasizes on the tools that are using various semantics. Previous papers focused on another part of the project around the transformation techniques [5]. This paper is structured as follows. Section 2 describes the CO-OPN specification language, the following section 3 introduces the SANDS environment, section 4 the prototyping methodology for distributed systems, while the description of the simulator is done in section 5, and section 6 presents the test set selection method. Concluding remarks can be found in section 7.

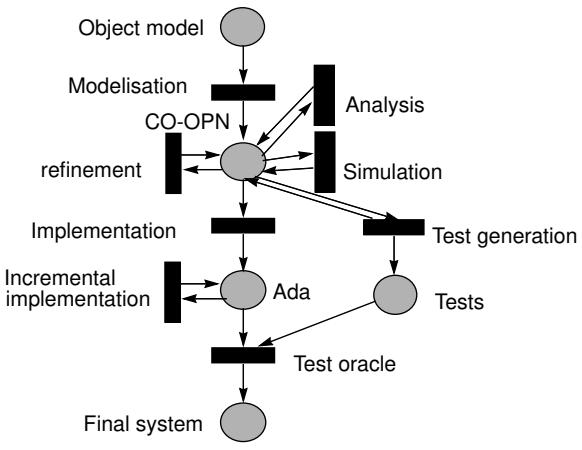


Fig. 1. Activities and formalisms within SANDS/CO-OPN

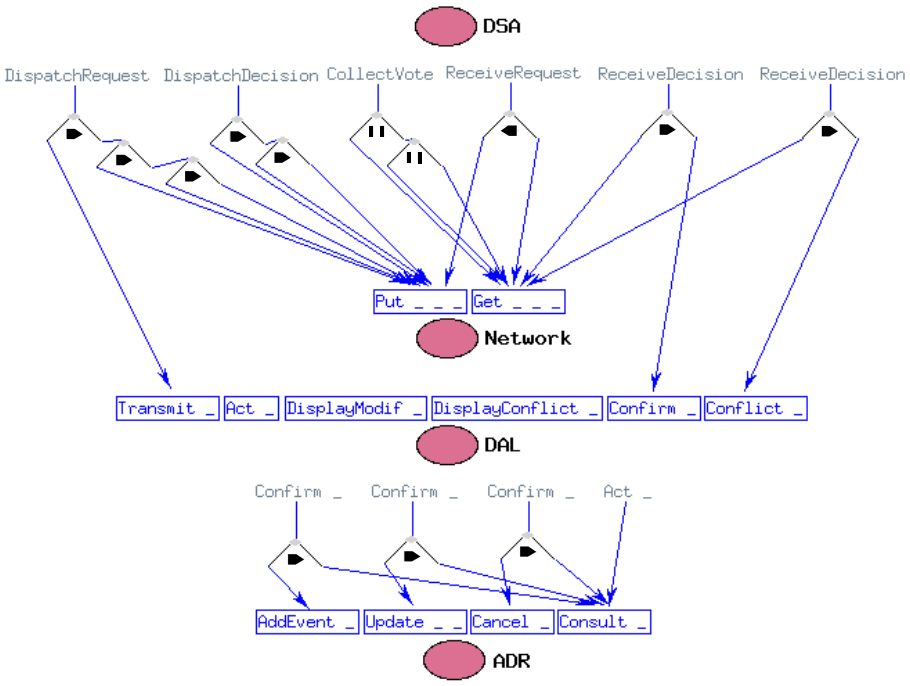
2 The specification language: CO-OPN

2.1 Basic ideas

The CO-OPN specification language is based on two elegant concepts, *Algebraic data types* and *Petri nets*, which are merged into a more general model *Algebraic nets* [22]. In order to express efficiently certain operations on large instances, the standard constructors on data types, structuring capabilities and the genericity principle (both for objects and algebraic data types) are also included in CO-OPN.

2.2 Concepts

The formal model is based on an approach in which encapsulation is ensured by methods and abstraction is supported by synchronization expressions between methods. An *object* is an algebraic net, with parameterized transitions as methods and the possible markings of the places as states.



We define a simple shared diary for members of a 'software engineering laboratory' with a replicated architecture. This diary allows the following services: Consultation, addition, cancellation and replacement of events. An event is simply composed of a day, a beginning and ending time and a comment line. Each site has four components for the diary of each member: the Abstract Document Representation (**ADR**), the Distributed Synchronization Algorithm (**DSA**), the

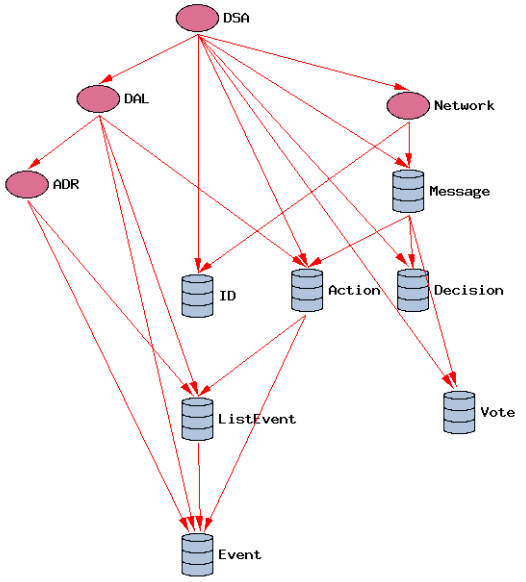
Fig. 2. Global view of the synchronizations in the control of a collaborative diary

According to our terminology, the events which represent the evolution of the net can be of two types: *methods* or *internal transitions*. The methods of an object are activated externally whilst internal transitions are activated locally, as soon as their conditions are validated. The events modify the state of the system and the symbolic links between the objects express synchronization constraints between the events of these objects. The constraints are described by a combination of sequential, simultaneous, alternative or empty synchronization operators.

In summary, our formalism is based on the following components:

- Algebraic abstract data types
- Petri nets
- Modularity and synchronizations

From a graphical point of view, a model can be seen at two different levels. The first level is an abstract view of the application which represents the CO-OPN objects with synchronizations between them (figure 2 and figure 3). The second level represents the algebraic Petri net within each object and contains places, transitions and methods with parameters (figure 4). A method is the entry point for synchronizations with other objects.



In this figure each oval represents an object - one Petri net - and each drum stands for an abstract data type module:

- **ADR** keeps all information concerning the diary definition.
- **DSA** ensures that the replicated information is consistent.
- **DAL** filters accesses to the **ADR**.
- **Network** is a model of the services expected from the real communication medium.

The abstract data type modules defined by this system are: **Message**, **ID**, **Action**, **Event**, **ListEvent**, **Decision** and **Vote**.

Fig. 3. Module enrichment relationships of the diary specification

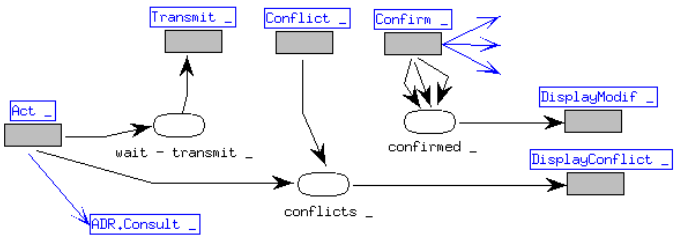
3 The environment: SANDS

The tools embedded in SANDS, represented in figure 1 with a data flow scheme, have been developed for covering all steps in the development of CO-OPN specifications. It includes a graphical editor which handles the objects and their organization, a compiler to check the specification in accordance with the semantics of the system, a monitor for simulating some part of the application, a code generator for evolutive prototypes generation and a test set generator.

3.1 The graphical editor

The modularity of the specification language is supported by the graphical editor. The user defines the global organization of the objects (figure 2 and figure 3), their external methods and the synchronization links between them. A graphical window is associated with each object (figure 4), allowing the visualization of its internal behavior which is described by an algebraic net.

All the rules of CO-OPN specifications (figure 5) are also handled by means of graphics. Clarity can be maintained by avoiding information overload. Algebraic specifications can be integrated by the editor and some operations can be executed on these.



This is the graphical representation of the behavior of object **DAL**:

- Method **Act** is offered to the client of the diary in order to validate an **action** - a modification - to the current list of events. If a conflict can be detected locally, the action is directly inserted into the **conflicts** place. Otherwise, the action is transmitted to the place **wait-transmit**, so that the **DSA** object may, by calling method **Transmit**, further validate it globally on the network.
- Method **Conflict** allows the **DSA** to signal actions that conflict with the other participants of the diary.
- Method **Confirm** allows the **DSA** to provide actions that have been globally validated on the network.
- Methods **DisplayModif** and **DisplayConflict** are to tell the graphical interface when new events are available for displaying.

Fig. 4. An algebraic Petri net: Internal view of object **DAL**

```
OBJECT DAL;
INTERFACE
  USE ADR, ListEvent, Event, Action;
  METHODS
    Act _            : action;
    DisplayModif _   : listevent;
    DisplayConflict _: action;
    Transmit _       : action;
    Confirm _        : action;
    Conflict _       : action;

BODY
  PLACES
    confirmed: listevent;
    conflicts: action;
    wait-transmit: action;
  AXIOMS
    ActOk :: Consistent(a,l)=true =>
                 Act a with Consult(l) : -> wait-transmit a;
    ActNotOk :: Consistent(a,l)=false =>
                 Act a with Consult(l) : -> conflicts a;
    ...
  WHERE
    a : action;
    l : listevent;
END DAL;
```

This is the textual specification corresponding to the graphical representation of figure 4. Not all method behaviors are listed here.

The syntax of an event (method or transition) axiom is:

```
AxiomName :: AxiomCondition =>
   EventNameAndArguments with AxiomSynchronization :
     AxiomPrecondition -> AxiomPostcondition ;
```

Fig. 5. Specification of object **DAL** (partial) and of method **Act**

3.2 The simulation phase based on operational semantics

CO-OPN specifications can be either simulated using the monitor or executed on a distributed net. Figure 8 describes the complete development process for this last possibility. In both instances, the CO-OPN code will have to be processed by a common front-end which performs lexical, syntactical and static semantic analysis.

3.2.1 A centralized view

If a simulation is desired, specifications will be translated into Prolog. In this case an abstract operational semantics is used to perform this transformation. By adopting a centralized view of the semantics of CO-OPN, we can exploit the nice features of Prolog unification and resolution in order to compute the behavior of a given specification. The resulting code can be loaded into the monitor to receive a highly interactive evaluation of the application.

3.2.2 A distributed view

From a behavioral point of view, automatically generated prototypes are used when it is necessary to validate the specified system in its actual environment, which can be a distributed system. Centralized techniques do not apply in this context and we designed a concrete operational semantics for this purpose. This semantics is more complex than the centralized one because all implicit object synchronizations during an event occurrence must be made explicit. There are strong similarities with the semantics of distributed Prolog. Moreover, the management of complex synchronizations, which are atomic from the operational point of view, is based on general nested transaction algorithms.

4 Prototyping on distributed systems

The idea is that the compiler is to be used to generate code for simplified CO-OPN systems: We assume that some restrictions, be it on data flows or free terms that cannot be instantiated, are satisfied.

4.1 The main principles

This section presents the new prototyping methodology for large concurrent systems modelled with CO-OPN. First, the source specifications are automatically translated into a main-stream object-oriented language, thus providing a portable and high-level initial implementation. The generated class hierarchy is designed so that the developer may then independently derive new sub-classes in order to make the prototype more efficient or to add functionalities.This process is performed incrementally in order to safely validate the modifications against the semantics of the specification. The resulting prototype can finally be considered as the end-user implementation of the specified software system. The originality of our scheme is that we exploit object-oriented programming techniques in the implementation of formal specifications in order to gain flexibility in the development process.

4.2 The Evolutive Prototyping Process

Our approach can be characterized by contributions on two levels: First, the class-based separation between the automatically generated and the hand-written code, and second, the use of prototype objects in a systematic software development process.

4.3 A Class-Based Decomposition

The most common application of the object-oriented paradigm consists in designing and implementing reusable and extensible software components [19]. Researchers in the field of computer simulation were the initiators of this trend [10]. The modeling power it provides does not have to be proven anymore, and has been exploited in numerous applications, also for the implementation of Petri nets, like in XSimNet [13]. The CO-OPN notion of object is represented as a class composed of places and exports procedures which implement the specified transitions and methods.

```
type Abstract_DAL is abstract new Root_Object with private;

procedure Act (O: access Abstract_DAL; a: in Action; Result: in out Status) is
  l: ListEvent := ListEvent_Prototype;
  Consult: Consult_Xid;
  ChosenAxiom: Act_Axiom_Name;
begin
  DeclareSync(Consult);
  ChooseAxiom((ActOk,ActNotOk),ChosenAxiom);
  case ChosenAxiom is
    when ActOk =>
      call(ADR,Consult,l);
      if Consistent(a,l)/=true then return; end if;
      put(O.wait_transmit,a);
    when ActNotOk =>
      call(ADR,Consult,l);
      if Consistent(a,l)/=false then return; end if;
      put(O.conflicts,a);
  end case;
  Result := success;
end Act;
```

This is what the generated code looks like:

- A declaration of the encapsulated class **`Abstract_Dal`**, which is the abstract view of object **`DAL`** at the programming language level. The complete definition is not given here.
- For each event (method or transition) of the specification, the definition of a procedure corresponding to all axioms of an event definition. Parameter **`Result`** tells if the event could be fired or not.

```
procedure Act (O: access Concrete_DAL; a: in Action; Result: in out Status) is
    l: ListEvent := ListEvent_Prototype;
    Consult: Consult_Xid;
begin
  DeclareSync(Consult);
  call(ADR,Consult,l);
  if Consistent(a,l)=true then
    put(O.wait_transmit,a);
  else
    put(O.conflicts,a);
  end if;
  Result := success;
end Act;
```

This hand-written version of **`Act`** belongs to a subclass of **`Abstract_DAL`** called **`Concrete_DAL`**. It optimizes away the non-deterministic choice of axiom to execute by exploiting the fact that the axiom conditions discriminate between the two situations to cover.

```
procedure Act (O: access Monitored_DAL; a: in Action; Result: in out Status) is
begin
  Act(Concrete_DAL(O),a,Result);
  if Result = success then
    highlight(O.Act_Graphical_Object);
  else
    reset(O.Act_Graphical_Object);
  end if;
end Act;
```

A developer (or a tool) could, by deriving a class **`Monitored_DAL`**, redefine the procedure **`Act`** to signal graphically on a customized display when method **`Act`** is fired. Notice that we reuse the inherited version of **`Act`**, thus preserving the correctness of the operation.

Fig. 6. Extracts of generated and prototyped Ada code for object **`DAL`** and method **`Act`** (cf. also figure 5)

The paradigm of object-orientation permits easily giving several implementations for a same abstraction. In usual implementation patterns there is one parent class with several equally important sibling implementations. We propose instead a hierarchy with a privileged implementation, the one generated automatically, which will serve as reference for ulterior hand-written class derivations. Figure 7 illustrates this organization: The abstract class contains, in the form of procedures, the translation of the Petri net behavior. By proper use of polymorphism, this class may export all its operations in an implementation independent manner. The generated concrete class furnishes a sample internal representation to be used by default. It does usually not redefine any of the inherited routines.

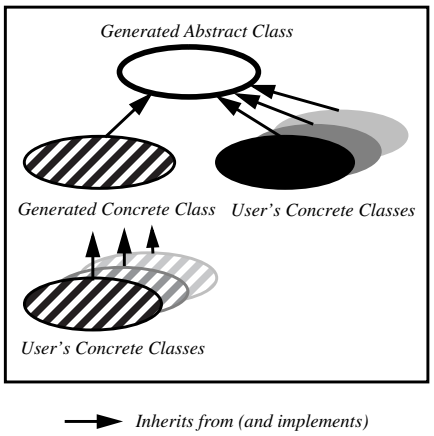


Fig. 7. Evolutive prototyping states of a class

Starting from this initial structure, the developer can provide his own range of sub-classes, where the internal representation is better suited to his own needs. Then, he can incrementally redefine the operations inherited from the abstract class, and optimize them for the chosen internal representation, or provide functionalities which cannot be specified with Petri nets. This process is depicted in figure 8:

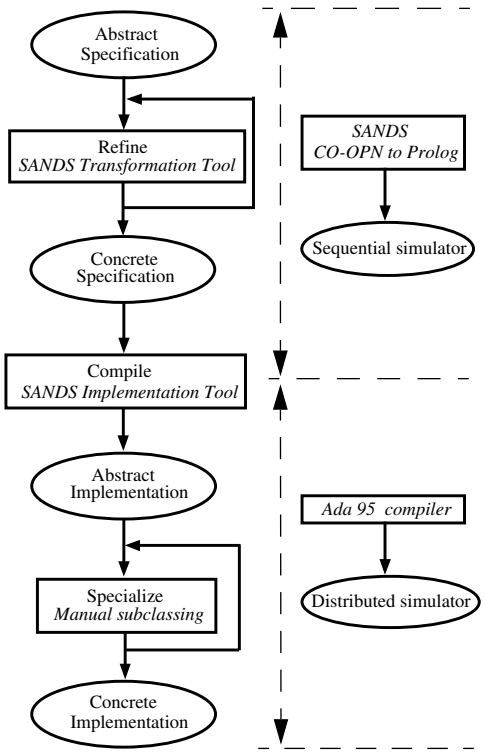


Fig. 8. Simulator development process

The evolutive simulator is active only in the lower half of figure 8, but we preferred to show how the methodology sticks in the global development process. One striking characteristic of this prototyping technique is that the model can be simulated at any moment: The slightest modification can be immediately integrated and tested, by means of direct evaluation, against deviation from the specifications. Another important feature which contributes to the safety of the method is that the automatically generated code is never modified: Instead, the developer derives new sub-classes which incorporate or replace the inherited code with hand-written extensions.

A related technique is the one of *mixed prototyping* [8], which is also based on the compilation of formal specifications, and where the generated code may be progressively replaced by hand-written modules. The drawback

with their method is that code reuse is very limited, which is unsatisfactory from a software engineering point of view, but also excludes some of the more interesting combinations we demonstrate here.

4.4 The Flexibility of Prototype Objects

In our prototyping scheme, we would like additional flexibility and dynamics in order to swap implementations without having to modify or recompile client modules. This is made possible through the use of *prototype objects*, also sometimes called *exemplars* [9], which is a natural programming technique in languages like Smalltalk and Self. Our use of object-orientation is related to the general notion of *open implementation* [17], where the objective is to allow the client to choose some implementation details of the modules it uses. This is most easily realized through *reflection* and *meta-object protocols*, see for instance [16], but may be simulated in conventional languages like C++ and Ada95 through the use of *creational patterns* [12] like prototype objects.

This trick, combined with dynamic binding, allows our compilation tool to generate an abstract class serving as unique interface for all the concrete implementations. Client modules do not know which of the concrete subclasses is actually used. The prototype objects are initialized at program start-up according to the developer's indications in an auxiliary configuration file. It is not the purpose of this paper to give any further details about the realization of this mechanism. Interested readers can consult [15].

5 The simulator

The reachability graph of the system is built during its simulation. When there is no simulation in process, the states graph can be explored. Different facilities are offered to perform this exploration.

The environment provides three main means for interaction: a step-by-step evolution, an evolution controlled by the synchronization expressions between objects in the system and a HML-like logic verifier by means of which paths verifying a specified formula may be built. This last feature is illustrated in section 6.2.

5.1 Type of evolution

In a *step-by-step* mode, the firability of the methods of the objects are tested in the current state of the system, whereas in a *synchronized* evolution, a constraint expression, involving events of objects and notions of sequentiality, parallelism and alternatives, is specified by the user.

The development of the *high-level reachability graph* depends on the choice a particular state from which its subsequent successors may be computed. The choice is left to the user or may be selected at random by the monitor. The choice of the state from which the reachability graph is developed can be chosen from the set of already-computed states. Therefore, the development of one branch of the graph can be abandoned for the benefit of another.

5.2 Abstract data type evaluation

During simulation, the user can also test the behavior of the abstract data type part of the system, i.e. he can evaluate a term expression where variables may occur. Equality between two expressions, possibly with free variables, can also be proved by using resolution techniques. Roughly speaking, the evaluation is performed through orientation of the axioms.

5.3 Generic marking

Due to the use of variables within the terms of the contents of places, a marking can be viewed as a generic marking, i.e. each instantiation of a free variable generates a particular marking. This ability also has an impact on the reachability graph (figure 9).

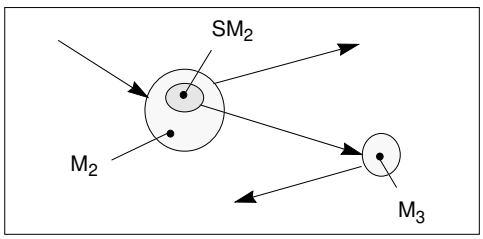


Fig. 9. The generic marking SM_2 is the sub-marking of M_2 ($SM_2 \subseteq M_2$) such that it enables the step to the generic marking M_3

5.4 Constraints on the evaluation of synchronizations

In some cases, a synchronization expression may produce a very large or even an infinite number of successors. If needed, some constraints can be specified in order to overcome this difficulty. Parameters such as depth, parallelism and width set a limit on, respectively, the number of steps from the current state, the number of transitions fired simultaneously in a single step, and the number of successors of a marking.

6 Test set generation

In our implementation theory [7], a correct implementation should be comparable to the specification from an observational point of view, it means that it should be observed independently from the states of the objects and from the way they are realized. We adopt bisimulation techniques in order to capture this intuition at the semantic level. In our testing theory [2] we give a theoretical framework for the test selection which implies the use of an adequate testing language: this problem needs an adaptation of the implementation theory. An interesting choice is HML logic which can express the expected behavior of an implementation from an observational point of view compatible with the bisimulation principles.

6.1 Construction of a Practicable Test Context and Reduction Hypotheses

We show in this section how to iteratively reduce a test set, in order to reach a test set that reflects the expected behaviors that are crucial for the validation of an implementation with respect to its specification. The definition of a practicable test context is the following: Given a specification *SP*. A test context $(H, T_{SP, H})_O$ is defined by a set of hypotheses H on a program under test P. A test set is practicable if it is pertinent and has a "reasonable" finite size. To simplify, $(H, T_{SP, H})_O$ is noted $(H, T)_O$ in the rest of the paper.

The selection of a pertinent test set $T_{SP, H}$ of "reasonable" size is made by successive refinements of an initial test context $(H_0, T^0)_O$ which has a pertinent test set T^0_{SP, H_0}, until the attainment of a practicable test context $(H, T)_O$:

$$(H_0, T^0)_O \leq ... \leq (H_i, T^i)_O \leq (H_j, T^j)_O \leq ... \leq (H, T)_O.$$

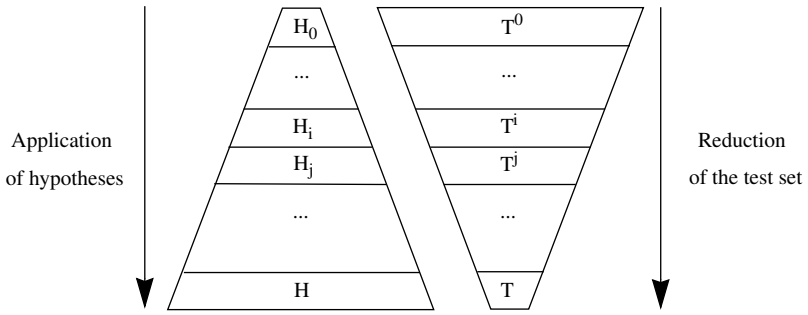


Fig. 10. Iterative refinement of the test context

At each step, the preorder refinement context $(H_i, T^i)_O \leq (H_j, T^j)_O$ is such that: The hypotheses H_j are stronger than the hypotheses H_i

- The test set T^j_{SP, H_j} is included in the test set T^i_{SP, H_i}
- If P satisfies H_j then $(H_j, T^j)_O$ does not detect more errors than $(H_i, T^i)_O$:
- If P satisfies H_j then $(H_j, T^j)_O$ detects as many errors than $(H_i, T^i)_O$:

Therefore, we have the properties that given two test contexts $(H_i, T^i)_O$ and $(H_j, T^j)_O$ such that $(H_i, T^i)_O \leq (H_j, T^j)_O$. If T^i_{SP, H_i} is pertinent then T^j_{SP, H_j} is pertinent.

A test is a formula with or without universally quantified variables. The reduction hypotheses act on tests with variables, replacing theses variables to reduce the formula complexity. Thus we define $Var(\phi)$: the set of variables of a formula ϕ.

6.1.1 The Uniformity Hypotheses

The uniformity hypotheses, which are very strong, stipulate that if a test of a formula ϕ, with $v \in Var(\phi)$, is successful for a given value of v, then it is successful for all possible values of v. For instance, in the case of a stack, if a test is successful for a stack which has a top element of 2, then it is successful for all possible values of the top element. Since the uniformity hypothesis is very strong, it is usually not applied alone to the sort under test, but to the sorts imported into the specification, which we suppose being already tested. Uniformity hypotheses can be combined with domain decomposition.

6.1.2 The Regularity Hypotheses

The regularity hypotheses stipulate that if a test of a formula ϕ, containing a term t, is successful for all terms t which have a complexity less or equal than a bound k, then it is successful for all possible complexities of t. For instance, in the case of an unbounded stack, if a test is successful for all stacks of size smaller or equal to 20, then it is successful for all possible stack sizes.

In our case, if a uniformity hypothesis or a regularity hypothesis is applied on a pertinent test set, then the new test set obtained is pertinent too. Note that those two hypotheses correspond (possibly in a more formal way) to usual test practices. We study the *HML* formula domain decomposition, which can automatically give us interesting subdomains of the method specification.

6.2 Test set generation

Here we present some examples applied to the sample specification. The operational techniques used for the test set selection are based on the use of valid and complete logic programming techniques which are able to control the resolution process. The purpose of such techniques is to find satisfiable goals corresponding to the valid temporal formulae, in a reasonable time (figure 11).

We operationally extract from the specification of **DAL** tests of the form < validity, HML formula > :

```
[coopntest]: selectprint ([tests(1), utest, valid]).

TEST(0):
    <TRUE,
     NOT <Act (Update (< DMY(1, 0, 1), HM(1, 1), HM(0, 0), [] >, < DMY(1, 2, 0),
HM(0, 2), HM(2, 0), [] >))>>

TEST(1):
    <TRUE,
     NOT <Act (AddEvent (< DMY(4, 0, 1), HM(0, 0), HM(0, 1), [] >))>>
...


TEST(24):
    <FALSE,
     <Update (< DMY(0, 0, 0), HM(1, 1), HM(1, 1), a'bs'sp'm'[] >, <DMY(1, 2, 0),
HM(0, 0), HM(1, 2), [] >)>>

TEST(25):
    <FALSE,
     <Cancel (< DMY(1, 0, 0), HM(0, 1), HM(1, 4), [] >)>>
```

This is an excerpt of the test set of depth level one regularity for object **DAL** with our uniformity hypotheses on variables. Method **Act**, which has a parameter of type action, is tested first. Its arguments are unfolded with respect to the following method and generator profiles:

```
Act _      : action;
Update _ _: event event -> action;
< _ _ _ _ >: date daytime daytime comment -> event;
DMY _ _ _ : nat nat nat -> date;
...
```

Fig. 11. Sample test set for object **DAL**

7 Conclusion

We have presented the characteristics of four main components of the SANDS environment which are: The language, the simulation tool, the prototype generator and the test set generator. All these tools are carefully built upon the different kinds of semantics that have been defined for CO-OPN, showing the necessity of developing various levels of semantics for formal languages.

8 Acknowledgments

This work has been supported partially by the Esprit Long Term Research Project 20072 “Design for Validation” (DeVa) with the financial support of the OFES (Office Fédéral pour l'Education et la Science), and by the Swiss National Science Foundation project “Formal Methods for Concurrent Systems".

9 References

[1] S. Barbey, M. Ammann, and A. Strohmeier. Open issues in testing object-oriented software. In Karol Frühauf, editor, *ECSQ '94 (European Conference on Software Quality)*, pages 257– 267, Basel, Switzerland, October 17-20 1994. vdf Hochschulverlag AG an der ETH Zürich. Also available as Technical Report (EPFL-DI-LGL No 94/45).

[2] S. Barbey, D. Buchs, and C. Péraire. A theory of specification-based testing for object-oriented software. Proceedings of EDCC2 (European Dependable Computing Conference), LNCS 1150, Springer Verlag, Taormina, Italy, October 1996, pp. 303-320, Also available as Technical Report (EPFL-DI-LGL No 96/163).

[3] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *IEEE Software Engineering Journal*, 6(6):387–405, November 1991.

[4] O. Biberstein and D. Buchs. Structured algebraic nets with object-orientation. In *Workshop on Object-Oriented Programming and Models of Concurrency '95*, Turin, June 1995.

[5] D. Buchs, M. Buffo, and J. Rolim. Developing parallel programs based on rewriting techniques. Compilers and tools for parallel processing. *IEEE/USP workshop on high performance computing* - WHPC'94, São Paulo (Brazil), March 28-30 1994.

[6] D. Buchs, O. Biberstein, M. Buffo, C. Buffard, J. Flumet, J. Hulaas, G. Di Marzo, and P. Racloz. November 1995. SANDS1.5/CO-OPN1.5: An Overview of the Language and its Supporting Tools. *Technical Report 95/133*, Department of Computer Science, Swiss Federal Institute of Technology, Lausanne.

[7] D. Buchs and N. Guelfi. A concurrent object-oriented Petri nets approach for system specification. In M. Silva, editor, *12th International Conference on Application and Theory of Petri Nets*, pages 432–454, Aarhus, Denmark, June 1991.

[8] C. Choppy and S. Kaplan. 1990. Mixing Abstract and Concrete Modules: Specification, Development and Prototyping. In *Proc. 12th Int. Conf. on Software Engineering* (Nice, France, March). 173-184.

[9] J.O. Coplien. 1992. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, Ma.

[10] O.J. Dahl and K. Nygaard. 1966. SIMULA - An ALGOL-Based Simulation Language. *Communications of the ACM*, 9(9): 671-678.

[11] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Ma.

[13] Å. Gustavson and A. Törn. 1994. XSimNet, a Tool in C++ for Executing Simulation Nets. In *Proc. of 1994 European Simulation Multiconference* (Barcelona, Spain, June 1-3), 146-150.

[14] J. Hulaas. An evolutive distributed algebraic petri nets simulator. In A. Javór, A. Lehmann, and I. Molnar, editors, *Modelling and Simulation 1996*, pages 348–352, Budapest, Hungary, June 2-6 1996. 10th European Simulation Multiconference ESM96, Society for Computer Simulation International.

[15] J. Hulaas. 1995. Introduction to Object-Oriented Mixed Prototyping with Algebraic Specifications. *Technical Report 95/155*, Department of Computer Science, Swiss Federal Institute of Technology, Lausanne (Oct.).

[16] Y. Ichisugi, S. Matsuoka, and A. Yonezawa. 1992. Rbc1: A Reflective Object-Oriented Concurrent Language without a Run-Time Kernel. In *Proc. Int. Workshop on New Models for Software Architecture and Protocols'92*; Reflection and Meta-Level Architecture. 24-35.

[17] G. Kiczales. 1996. Beyond the Black Box: Open Implementation. *IEEE Software*, Vol. 13, no. 1 (Jan.): 8-11.

[18] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[19] B. Meyer. 1988. *Object-oriented Software Construction*. Prentice-Hall International (UK) Ltd, Hemel Hempstead.

[20] B. Marre. *Sélection automatique de jeux de tests à partir de spécifications algébriques en utilisant la programmation logique*. PhD thesis, LRI, Université de Paris XI, Orsay, France, January 1991.

[21] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[22] W. Reisig. 1991. Petri Nets and Algebraic Specifications. In *Theoretical Computer Science*, Vol. 80, Elsevier, 1-34.

[23] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[24] E. J. Weyuker. The oracle assumption of program testing. In *13th International Conference on System Sciences*, pages 44–49, Hawaii, USA, 1980.