ISSN 0105-8517

Third Workshop on Modelling of Objects, Components, and Agents

Aarhus, Denmark, October 11-13, 2004

Daniel Moldt (Ed.)

DAIMI PB - 571
October 2004

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
IT-Parken, Aabogade 34
DK-8200 Aarhus N, Denmark

Preface

This booklet contains the proceedings of the third international workshop on "Modelling of Objects, Components, and Agents" (MOCA'04), October 11-13, 2004. The workshop is organized by the "Coloured Petri Net" Group at the University of Aarhus, Denmark and the "Theoretical Foundations of Computer Science" Group at the University of Hamburg, Germany. The homepage of the workshop is: http://www.daimi.au.dk/CPnets/workshop04/

Objects, components, and agents are fundamental concepts often found in the modelling of systems. Even though they are used intensively in software engineering, the relations and potential of mutual enhancements between Petri-net modelling and the three paradigms have not been finally covered. The intention of this workshop is to bring together research and application to have a lively mutual exchange of ideas, view points, knowledge, and experience. The submitted papers were evaluated by a programme committee, which was supported by several members of their groups resulting in three to five reviews per submitted paper. The programme commitee consists of:

Wil van der Aalst	The Netherlands
Rémi Bastide	France
Didier Buchs	Switzerland
Piotr Chrzastowski-Wachtel	Poland
Jose-Manuel Colom	Spain
Jörg Desel	Germany
Giuliana Franceschinis	Italy
Tom Holvoet	Belgium
Nisse Husberg	Finland
Jens Bæk Jørgensen	Denmark
Ekkart Kindler	Germany
Gabriele Kotsis	Austria
Fabrice Kordon	France
Charles Lakos	Australia
Rainer Mackenthun	Germany
Daniel Moldt (Chair)	Germany
Francisco José Camargo Santacruz	México
Rüdiger Valk	Germany
Tomas Vojnar	Czech Republic
Wlodek M. Zuberek	Canada

The programme committee has accepted 12 papers for presentation. They tackle the concepts of objects, components, and agents from different perspectives. Formal as well as application aspects demonstrate how wide the range can be within which Petri nets can be used and illustrate at the same time that there is a tendency to use more high-level concepts for the analysis and design of Petri-net based models.

Daniel Moldt

Table of Contents

Unifying The Integration Of Real Time Computing Systems - An Abstract Machine To Specify Required Constraints

Rainer Mackenthun [*]

Abstract

This paper presents an abstract machine approach to specify required constraints. The specification is formal and contains functional and temporal system requirements. This approach is of special importance for integrators of Real Time Computing Systems, since they have to specify the required constraints in such a way, that it can be understood by developers and implementators in different domains, in different companies in different countries.

1 Introduction

Real Time Computing Systems (RTCS) have an increasing influence on any body's life. One of the major challenges is to integrate systems from different areas. A lot of effort has been invested in unifying the integration by standardizations of modeling languages, methodologies and processes. As the complexity of these integrated system explodes, it becomes obvious that the investments where mainly in specifying the solution design and the implementation of service platforms. The specifications of the required constraints is still far from standardization. This means, in an integration scenario, we are not able to formulate even important constraints in a precise way and we are not able to support the validation of RTCS sufficiently. A situation that should be changed, since integrated RTCS more and more conquer the area of dependable systems. We present an approach that could be a milestone on the way to a formal and therefore precise specification of required constraints formulated in the international language of mathematics.

The first half of this paper (section 2 to 4) describes the problem in detail, the second half (section 5 to 8) presents the solution.

2 RTCS Integration: Processes and Unified Methodologies

To explain the range of our work, we show the integration process in the context of systems evolution in figure 1. For a moment, let us assume that there exists only a single integration process. The integration process exchanges information with several development processes, integrates them and if the integration was successful give them to the implementation processes. This first integration is often called virtual integration. The result of the virtual integration is the specification.

The results of the implementation processes is running back to the integration process. This integration is often called real integration. The results of the real integration can be tested against the specification. These test are called conformance tests.

[*]Rainer Mackenthun, Fraunhofer ISST, Berlin, Germany, Rainer.Mackenthun@isst.fraunhofer.de

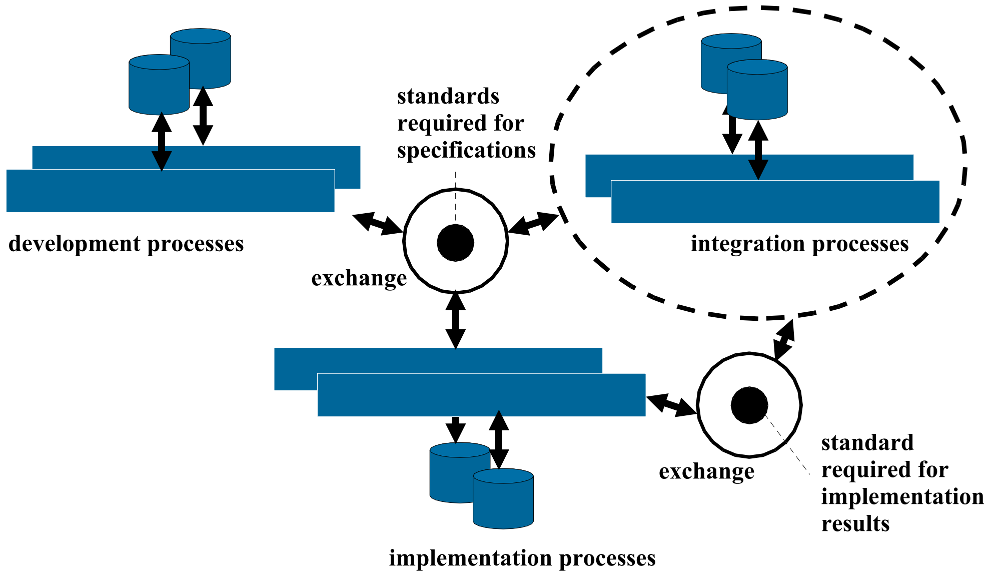


Figure 1: System Evolution Process

If there would be only one integration process, the modelling language used in this process could be the common language. As there are more than one, the standard modelling language must be a common agreement. This is a time consuming work, but the benefit is high: In principle, all developers can work with all integrators and all integrators can work with all implementators.

We conclude, that one of the main task for unifying the integration of RTCS is the definition of a standard description language of the specification and a standard for the implementation results.

We would like to go a step further an also unify the integration methodology. This requires to standardize not only the specification information, consisting of the description of the required constraints, the designed solutions and the implemented service platforms. In addition, the results of assisting activities should be described in a standardized way and the methodologies, how to come from one result to another. Figure 2 shows the results in our approach. The diagonal from the upper left to the lower right rectangle shows the results that belongs to the specification.

The other results are assisting description. Top down, the lines contain the constraint model, the design model and the implementation model. From the left to the right the columns contain the requirements, the solutions and the service platform. Our approach is different from the traditional computer science approach, as shown in figure 3. They defined three descriptions: requirements specification, design and implementation. Besides those, who just use this structure of results, there are still some people who belief in the existence of a solution generator. There belief is given by two axioms:

1. If we have a lot of experience in implementation, we can make such good design model, that we can generate implementation models from the design models.
2. If we have a lot of experience in design, we can make such good requirements models, that we can generate design models from these requirements models.

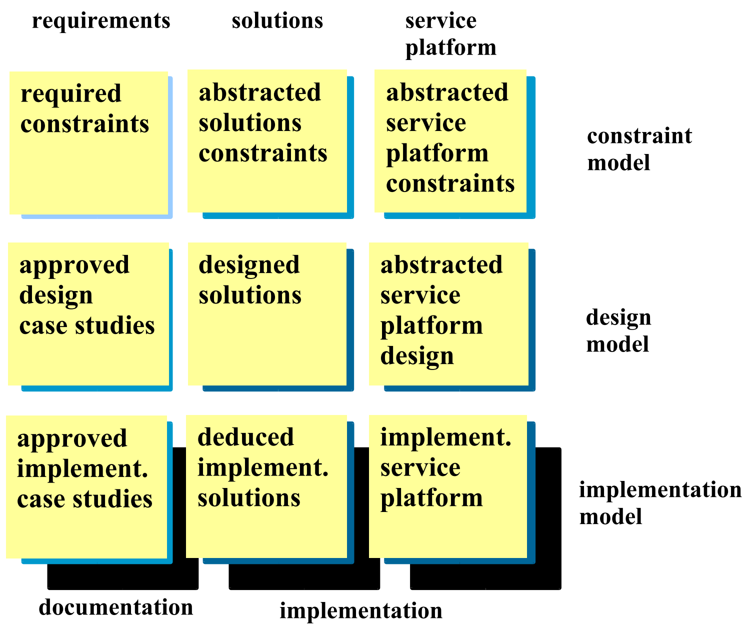

requirements
solutions
service platform
required constraints
abstracted solutions constraints
abstracted service platform constraints
constraint model
approved design case studies
designed solutions
abstracted service platform design
design model
approved implement. case studies
deduced implement. solutions
implement. service platform
implementation model
documentation
implementation

Figure 2: Results of the Integration

The final conclusion is, that one day we are so experienced, that we can generate new solutions from good requirements models. Philosophically seen, this is the acceptance of Aristoteles' induction axiom, that experience leads to stable abstract axioms from which the solutions can be deduced. Experience proves axioms.

We follow Karl R. Popper [Popper 34] who has argued against this belief and we are in line with many people in industry, with whom we discussed intensively in the past.

To analyse the belief in more detail, we switch to our approach and show the belief in our result structure (see figure 4A). The belief is, that we can come from the implementation results at the bottom to the specification results in the the diagonal by Aristoteles' induction. The rest, going upward from the specification is just abstraction of details from the solution design and the implementation of the service platform.

When we discuss with solution engineers in industry, they tell us, that we cannot find new design solutions for a concrete problem by induction on old implementations. This would mean: No steady experience field, no pure inductive learning, no generator for new solutions.

Our approach for requirements specification will introduce the requirements specification as the specification of that constraints, the integrator requires the system to do and that he wants to validate. Using Aristoteles' induction would mean that human beings eventually have experience of all the requirements, human beings can have on that system. This is against all our experience with human beings. If the believers are wrong, and we are right this would mean: no steady experience field, no pure inductive learning, no generators for new solutions.

To avoid confusion: we do not refuse generation and configuration as instrument to produce approved part of the system, that have only limited interaction with other parts of the system like subsystems, solution components and implementation components. We just argue for the use of creativity as the efficient instrument to bring innovations into a system. In the contrary, we encourage the use of generation and configuration so that the creative

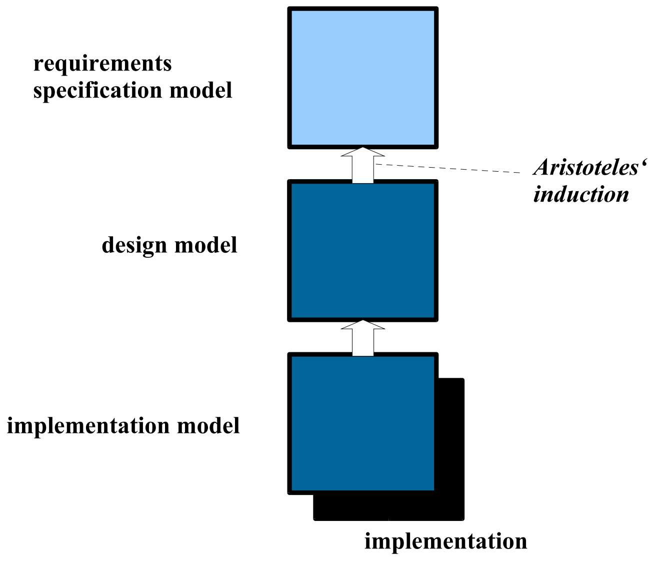


Figure 3: Traditional Computer Science Approach And The Generation Dream

resources in the system evolution process can concentrate on building dependable innovative solutions. We ourselves work in the area of domain engineering (saving the approve parts) and product lines engineering (choosing common parts in a product line). Both engineering approaches produce results, capable for generation and configuration. We think that the domain engineering is the area to use Aristoteles' induction.

We will no longer discuss this subject in this paper but offer our research results as an alternative approach. Let the industrial practice prove the hypotheses.

So we just remove Aristoteles' induction from our methodologies (see figure4B) and moves it to the domain engineering dimension that is not shown in the figure. We see innovations in the specification results as products of creativity. We propose to start with the requirements specification. As the requirements are on the abstract constraint level they should be validated via Popper's deduction (see figure4C). The requirements are deduced to design and implementation level in shape of case studies to check, if the constraint is implementable. This leads to a falsification or an approval of the constraint as requirements. Note, that the directed arcs in the figure describe the sequence of finishing the results. The black arcs of Popper's deduction seem to show in the wrong direction. But anyone who knows about Popper's theory is aware, that an abstract result is not established, until it is sufficient approved by concrete case studies. Even though we have to start with the abstract result and then deduce the concrete case studies, the ending of the activities is the other way round.

While the requirements specification is worked out, the service platform should be described. At a certain point in time, when the requirements specification exists and we have enough information from the service platform specification, we can start the creative process of solution design. The implementation on the solution can be done by Aristoteles' deduction, as there exist enough information from the implementation case studies and the service platform implementation to deduce solution implementations from the more abstract designed solutions.

We still use the abstraction from details for the other three results (grey arcs) and end

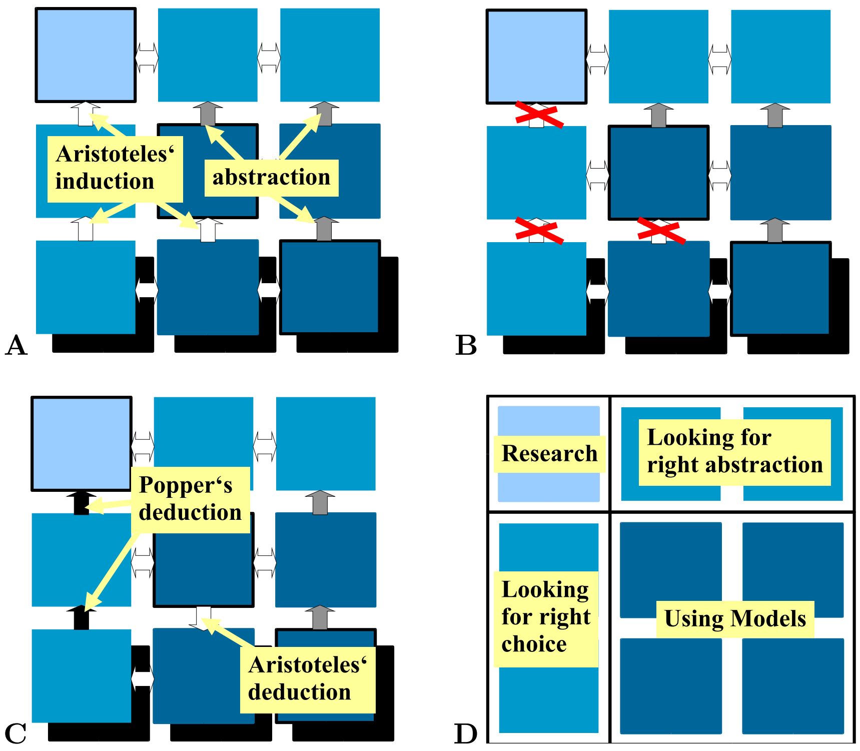


Figure 4: Integration Results: A) The Generation Dream B) Dream Destruction C) Our Proposal D) Industrial Practice

up with three models of the system that could be validated : the implementation model, the design model and the constraints model.

When we assess todays situation in industry, we come to figure 4D. To our knowledge at least certain parts of industry are doing quite well using models for design and implementation of solutions and service platform. The AUTOSAR initiative in the area of automotive electronic networks is a good example for this (cf. [Hardt, Feldo 04]). But the abstraction of a useful constraint model and the choice of useful case studies is still an open and heavily discussed question.

We offer as solution an approach for requirements specification that gives orientation for the choice of useful case studies and an abstraction level for constraint models.

3 Integrating The Object and Information World Solutions

We use requirements specification to give the integrator of a system an instrument to describe what he wants the system to do. We should be aware of two different philosophies that exists in the world of computing solutions. We call them *object world solutions* and *information world solutions*. The integrator can be in charge to integrate RTCS from both worlds.

As shown in figure 5 the requirements description should be unique for both worlds, since the worlds cannot be separated for the existence of hybrid solutions. Hybrid solutions e.g. start in one world and end in the other. The changeover between these worlds is organized via gateways.

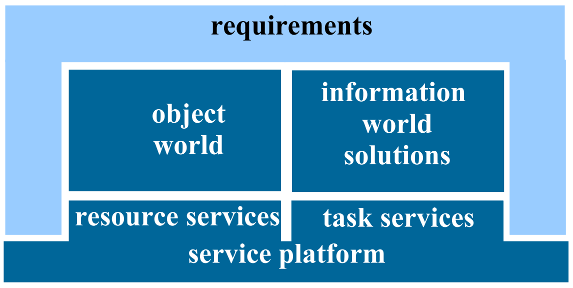


Figure 5: Unified Requirements Model

The information world solutions are influenced by the object-oriented paradigm. The solutions of this world have been influenced by the need for more features. The need for computer performance has always been satisfied by the hardware. So the solutions do not not really care about hardware performance. Performance just exists in a sufficient amount. Services in this world mean processing or task services. Send a task to the processing service and it will be processed in time. The axiom of (nearly) unlimited resources is fundamental to this world.

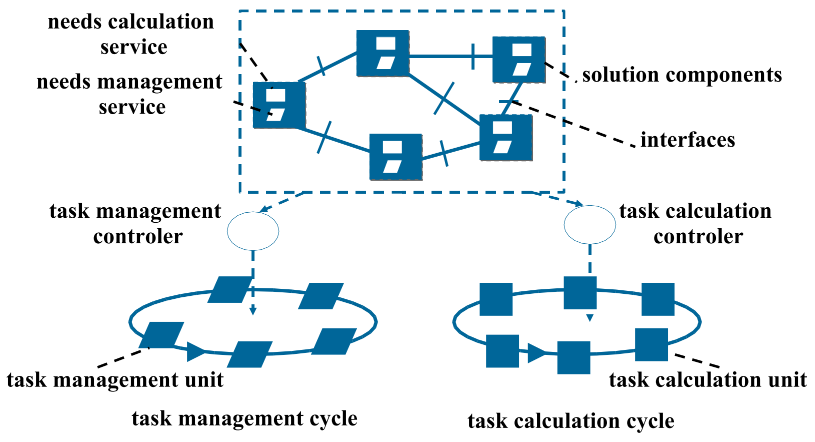
The principles of the information world solutions are shown in 6. Part A show the architecture, part B an example run.

Software solutions components are defined. If a solution component is instantiated to a solution task it needs a management service that, for instance, brings the task to the right execution place and a calculation service, that calculates the contents of the task. Solution components can communicate with each other via interfaces. So during the run, the tasks have to be synchronized to communicate with each other.

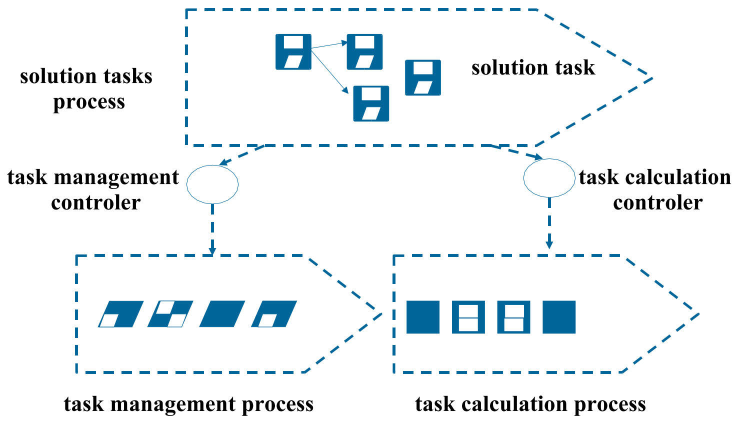
The task management is symbolized by the task management cycle. It contains task management units that are processes cyclically. During the run the task management units contain tasks to process. The task calculation works in a similar way. The task management controller is responsible to assign task to task management units, the task calculation controller is responsible to assign task to task calculation services.

In the first instance, specifying of temporal system requirements seems to be of no importance for this world. Unfortunately, the information world solutions are traditionally not used for dependable system, since they are not very reliable. The main problem is that for the high complexity, the synchronisation of tasks is not always successful. For the high complexity of the programs and therefore the tremendous amounts of possible execution situation this cannot always be checked sufficiently. For this purpose the definition of deadlines is of great importance. For instance if a system is able to calculate a number of tasks in half of the reaction time needed, one can put the deadline to this point in time. If it is not finished, you assume that the situation is deadlocked and the system may choose an alternative calculation. This is no guarantee for success but improves the dependability a lot.

In contrary to the information world, the object world has taken concepts from the real world like distribution, services that encapsulate the use of limited local resources or communication resources and resource controllers, that guarantee the avoidance of resource conflicts. The principles of this world are shown in figure 7. The axiom of limited resources is fundamental for the solutions in this world.



A



B

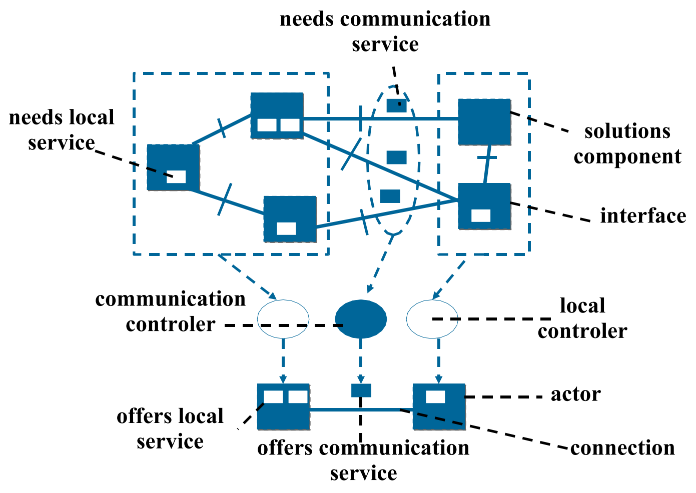
Figure 6: Architecture (A) and Run (B) of the Object World

Software solution components are defined. If a solution component is instantiated to a solution task it may need local services, e.g. to access a shared memory resource. Solution components can communicate with each other via interfaces. If a communication takes place, a communication service is needed to use a shared communication resource.

The local controller is responsible for the coordination of local service accesses. The communication controller is responsible for the coordination of communication service accesses. Software solution components are assigned to controllers, that are responsible for their service accesses. The location of a software component is identical to the location of the local controller, the software component is assigned to. Location are abstracted by actors that represent distributed hardware components.

To compare both approaches, in the information world the fast computing of many features play an important role. Distribution of task on hardware is abstracted in the management and calculation cycle. The model of runs should focus on task synchronization. In the object world limited resources and - for the limited communication services - distribution is important. Models of a run must respect this.

A



B

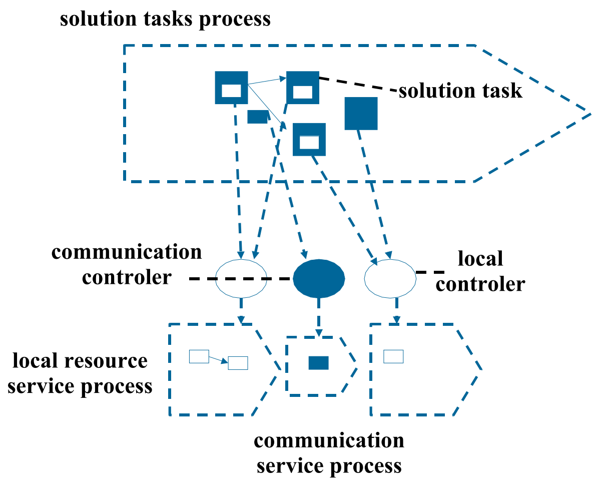


Figure 7: Architecture (A) and Run (B) of the Object World

4 Benefits Of Our Formal Requirements Specification

We focus on a requirements specification for the integration process. The requirements specification has to describe what the integrator wants the system to do and what he wants to check. What kind of properties could this be? For instance the integrator is interested that components that are build by different developers meet the deadline in every execution situation. Another property might be that hybrid solution meets deadlines when a changeover to the other world is necessary.

We would like to stress, that we do not try to achieve the goal of a formal correctness proof of a RTCS via the requirements specification. If properties can be proved, this should be done, but proving correctness is not the major objective. The requirements engineer in the integration process should get an instrument to describe important features of the system in a precise and understandable way. The important features we call the missions of the RTCS. As we use an abstract machine approach we call the executable requirements specification *abstract requirements machine (ARM)*.

The benefits of the ARM are twofold. At first they can be used to improve the system evolution process. Secondly they can be used to improve the system by integrating the ARM at runtime to improve the dependability or the security of the system.

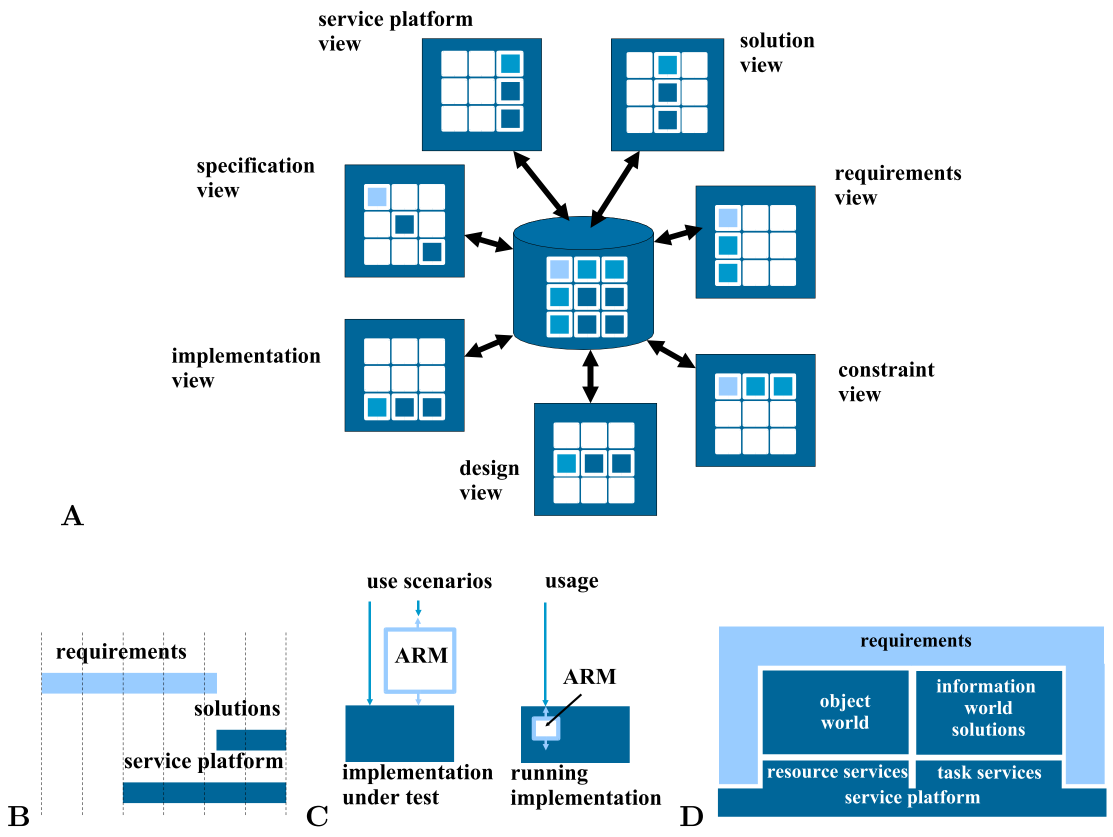


Figure 8: Benefits: A) Improving Integration Views B) Improving Evolution Process C) Improving System D) Improving Security of the System

Figure 8A shows that formal requirements specification would improve any view on the system integration model. We just mention those views, where the requirements specification has a direct influence. The specification view gets an international and easy understandable description of the system requirements. The constraint view is completed so that it can be used for validation and the requirements view is completed by adding the most abstract component which subsumes a lot of information that are currently redundantly contained in different requirements specification on an inconsistent abstraction level.

Figure 8B shows that a unified description of requirements is a major improvement for the early phases of the evloution process of the specification. This avoids conflict about subjects at the end of the process that could have been discussed at the beginning.

The right side of figure 8C shows the improvement of the system. The left side shows the usage of the ARM for testing the implementation. The right side shows the integration at runtime. For instance, the ARM could take the task of checking the deadline and to enable an alternative calculation.

As shown in Figure 8D the ARM can also be used to check the system for prohibited access scenario on an abstract level and therefore can be used to increase security of the system.

5 The Abstract Machine Approach For Constraint Models

Our solution for a constraint model uses abstract machines. These machine simulate the behaviour of our system on constraint level. We have chosen this approach for several reasons.

The number of system requirements is enormous. The most critical thing is, that these requirements are depending on the run of the system. This leads to a an enormous number of constraints even though we restrict to missions and makes it impossible to describe the requirements as rules. These direct approaches are rejected for their complexity, that must be managed.

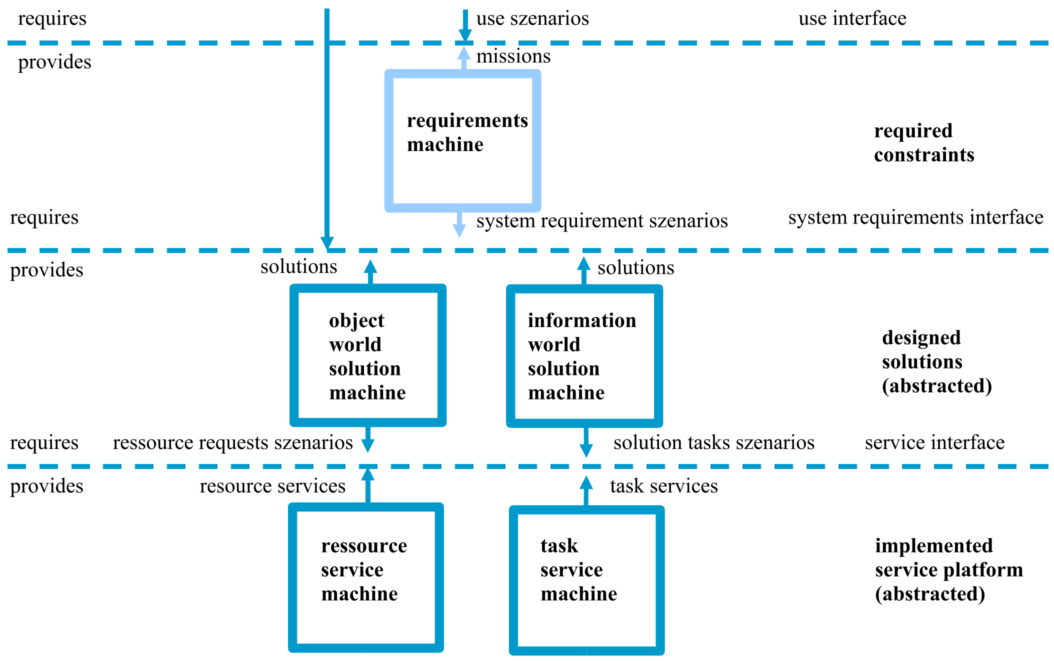

requires
provides
use szenarios
missions
use interface
requirements machine
required constraints
requires
system requirement szenarios
system requirements interface
provides
solutions
solutions
object world solution machine
information world solution machine
designed solutions (abstracted)
requires
ressource requests szenarios
solution tasks szenarios
service interface
provides
resource services
task services
ressource service machine
task service machine
implemented service platform (abstracted)

Figure 9: The Abstract Machine Approach For The Constraint Model

Alternative to the rule approaches there exists requirements specification that describe requirements scenarios implicitly . Timed and Coloured Petri nets, temporal logic and several other techniques are used for this purpose (see for instance [Bellini et. al. 00],[Heitmeyer, Mandrioli 95]). The scenarios are the partial ordered runs of specified executable models. Unfortunately there exists no sufficient techniques that is already available for requirements specification of real world RTCS in industry.

From our point of view, the main problems are, that the formal requirements specification is designed under the major objective of proving correctness and that the scenarios are described implicitly. This makes these approaches hard to use. In our approach the partial ordered runs are explicitly contained and manipulated. Our point of view towards correctness proofs is already mentioned above.

As we see before, we need the ARM as an instrument to formulate required constraints for the system, which are the abstract starting points - the theories - for Popper's deduction methodology. We use the formalization for the reason of preciseness and general understandability. So again, our objectives of the formalism is, that the requirements are simple to formulate and simple to understand by human beings without misunderstanding that are founded in an unprecise semantics of the description language.

So it seems to be like the “egg of Columbus” . We solved the problem, by giving up an assumed necessity: the prove of correctness. This does not mean that no property can be proved. But this is not the major objective.

Figure 9 shows the principle idea of abstract machines for the constraint level. You find 5 machines. On top we see the ARM that is presented in this paper. A certain use scenario is send to the requirements machine. This machine generates a system requirements scenario.

Later on, during validation the ARM is synchronously executed with the other four machines. There are two solution machines, one for the object world and one for the information world. The use scenario is also send to the two solution machines. These machines generate resource request and solution task scenarios. The resource service and the task service machine are as well executed synchronously. Analysers may validate the model at the shown interfaces.

Note that the solution and service machines are abstractions of other activities in the integration process as mention above. What kind of language should be chosen for the models? For the object world, we are working on a variant of the ARM, for the information world we see a lot of connections to the work in [Valk 98][Maier, Moldt 01].

A pragmatic approach could be to use - for the solution and service platform - the implementation instead of the models, as long as the models do not exist. The runnable implementation of solutions and the service platform can be included via monitoring. A precondition is that the implementations can be triggered in such a way, that a synchronous execution is possible.

6 The Abstract Requirements Machine

The abstract requirements machine is shown in figure 10. It is defined as a coloured Petri net (cf. [Jensen 97]). The details are explained in this section.

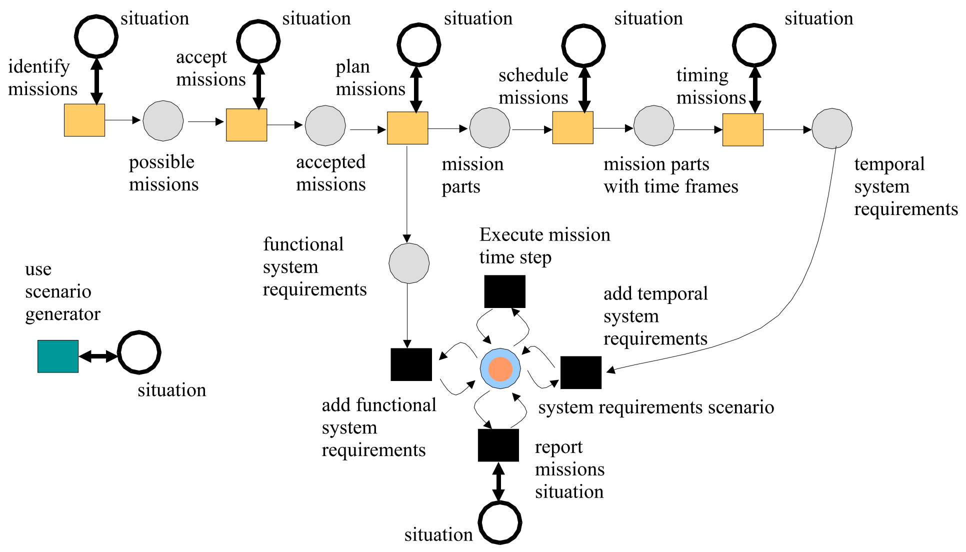

situation
identify missions
situation
accept missions
situation
plan missions
situation
schedule missions
situation
timing missions
possible missions
accepted missions
mission parts
mission parts with time frames
temporal system requirements
functional system requirements
Execute mission time step
add temporal system requirements
use scenario generator
situation
add functional system requirements
system requirements scenario
report missions situation
situation

Figure 10: The Abstract Requirements Machine

A main concept of our work is the *mission*. A mission of a RTCS is one of the important functions, a RTSC has to perform successfully. An airbag system has to perform the firing of an airbag in time if needed. The mission is the function the requirements engineer concentrates on.

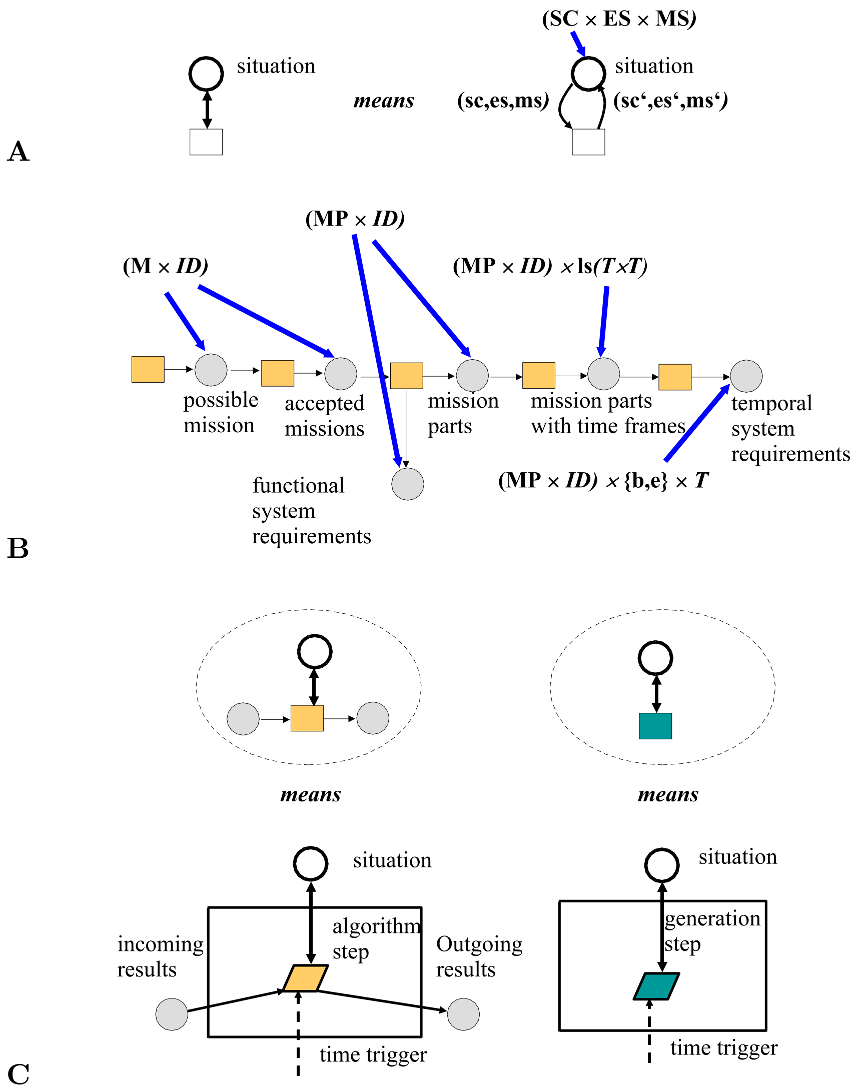


Figure 11: Specifiying the ARM: A) Fusion Place for the *Situation* B) Coloured Places for the Results C) Transitions as Mission Definition Transition (left) and the Scenario Generation Transition (right)

In a system, a lot of other work has to be done. We call portions of this other work *jobs*. In our requirements models, jobs are only considered by reserving time frames for them. Our approach meets the opinion of requirements engineers that the structuring of the requirements specification is different to the structuring of the designed solutions. On solution level, components are structured towards a design architecture e.g. to reduce the impact of a change. On requirements level the system activities are grouped along the missions and e.g. the missions are structured in such a way, that responsibilities for failures can clearly be investigated.

The ARM generates mission situations for use case scenarios. The mission situation are equal to the requirements scenarios and describe functional and temporal requirements constraints on the system.

Figure 11 shows the concepts, that have to be specified to generate the required constraints. The concepts are introduced and applied in the next section.

A) The situation is represented by a token on the bold rimmed fusion place *situation.* The token contains a value of a composed data type and behaves like a variable. The data type has to be specified by defining the scenarios type (data type SC), the execution situation type (data type ES), and the mission situation (data type MS). The situation type is the composed type: (data type $SI = SC \times ES \times MS$). The composed data type is the token colour of the fusion place *situation.*

B) The results that are exchanged between the yellow *mission definition transitions* are values of data type that are defined by the help of the following sets: the missions M, decision points DP, mission activities MA and mission parts $MP \subset M \times (DP \times MA \times DP)$ that are mission activities with the surrounding decision points. To define the token colours we further need an index set ID for the current instance of a mission, a set of points in time T that is equal to the set of natural numbers and a set $\{b, e\}$ that represent the two elements *begin* and *end* of an interval. The figure shows the token colours of the places of the light rimmed grey places.

C) The yellow *mission definition transitions* represent executable descriptions of an algorithm that is executed during the firing of the transition. The figure indicates that the description must offer an algorithm that can be time triggered. Every time the transition is fired, one time step of the algorithm is executed. The following specification activities have to be performed: Specifying the mission identification (result as transition *identify missions*) means describing scenarios in which a mission could be started. Specifying the missions acceptance (transition *accept missions*) means restricting the number of missions that are accepted in dependence of the current mission situation. For instance it might not be useful to start an airbag calculation, if another airbag calculation is already running. Specifying the mission planning (transition *plan missions*) means to describe which mission part is planned in dependence on the situation. Not all mission parts are planned at once. The mission planning can still decide about the next mission parts, when the mission is already running in another part. Specifying the mission scheduling (transition *schedule missions*) means to describe the assignment of time frame to a mission part in dependence on the situation. Here the reservation of certain time frames for jobs must be taken into account. Specifying the mission timing (transition *timing missions*) gives detailed time frame in which the mission parts will be executed in dependence of the situation. The green *use scenario generator* transition is similar to the yellow transition but does not send results but only changes the scenario part of the situation variable. Specifiying the *use scenario generator transition* means to describe how the use scenarios variable changes over the time.

The abstract machine will execute the use scenario generator transition and the mission definition transitions as specified and will build up the mission situation that represent the system requirements scenarios. The black *mission composition transitions* are automatically defined from the specified information. The missions situation is build up as a partial order of mission parts rsp. functional requirements (*add functional system requirements*) with assigned time interval attributes (*add temporal system requirements*). The current mission situation is updated by a clock transition (*execute mission time step*). The current mission situation is reported by a reporter transition (*report mission situation*) that updates the mission situation part of the situation variable.

We define modi for the abstract machine runs e.g. non-deterministic modus, where the values of the execution situation are changed in a non-deterministic way, test modus, where the values of the execution situation are changed as described in a test sequence, or run modus, where the values of the execution situation are monitored in the running systems.

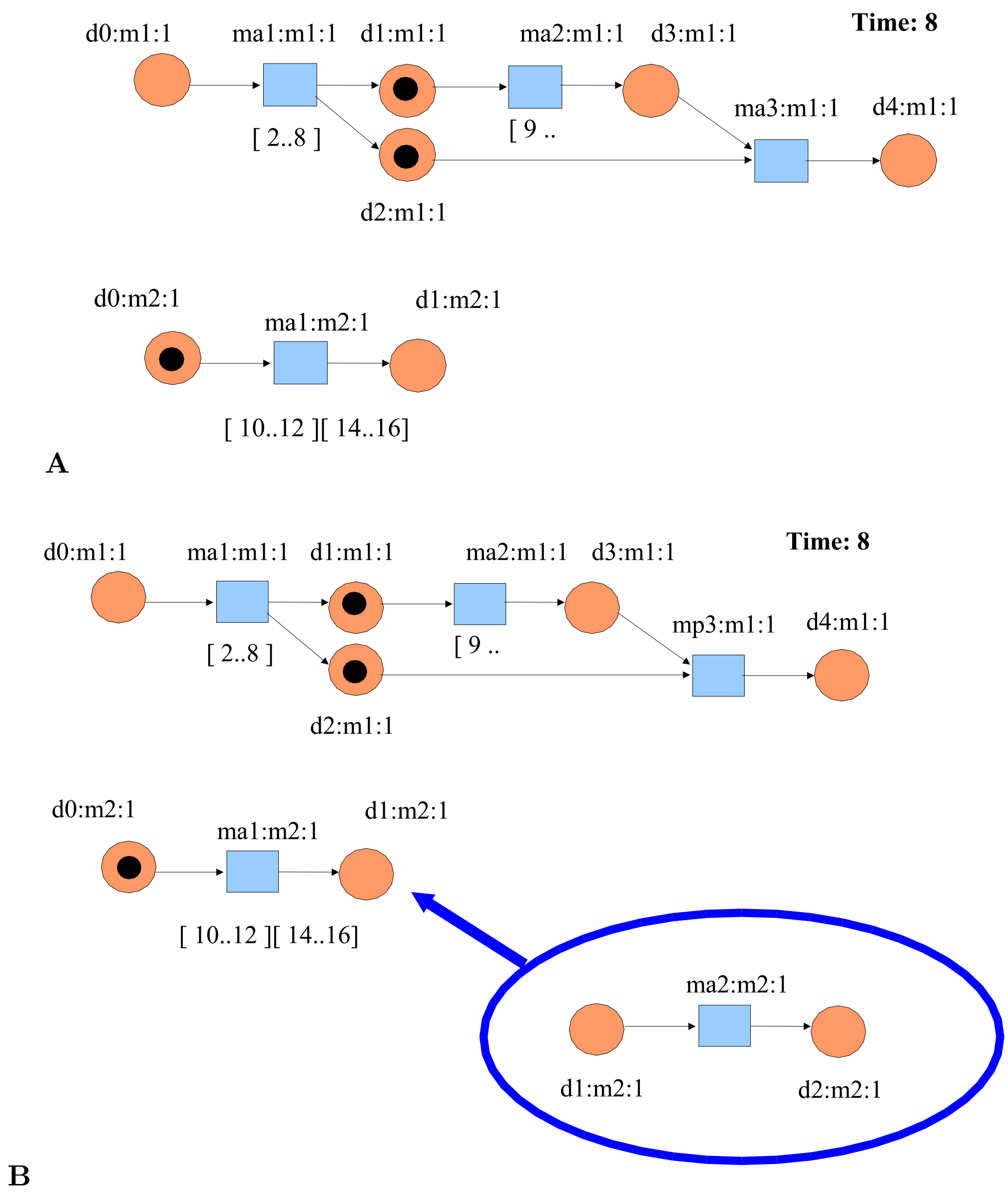
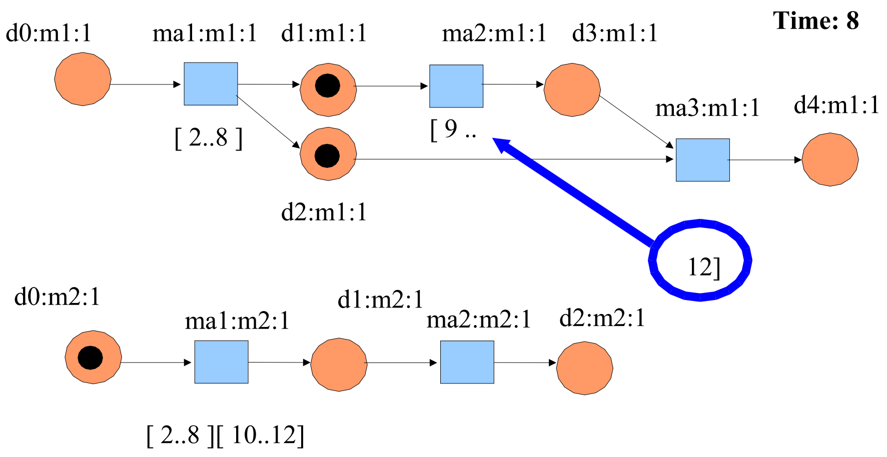


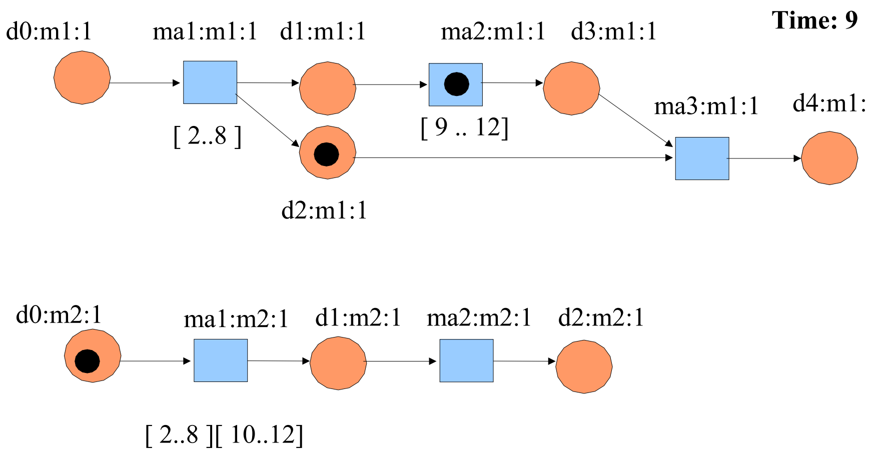
Figure 12: Mission Composition Transition: A) An Extended Causal Net on Place *System Requirements Scenario* B) An Effect Of Transition *Add Functional System Requirements*

Figure 12 and 13 show for an example of the mission situation rsp. the system requirements scenario how the black *mission composition transitions* effect the mission situation.

The mission situation at time 8 is shown in A). You see two missions *m1:1* (first instance of mission *m1) and* m2:1 (first instance of mission *m2*). Mission *m1:1* is consisting of partially ordered mission parts. Before, between and after mission activities are decision points. Some of the mission activities have assigned time frames. Mission activity *ma2:m1:1* has not yet got the time frames completely, but only the begin time stamp. The *mission state* is shown by the black token, also known as a cut. Figure B) shows an action of transition *add functional requirements*. Mission *m2:1* is extended by mission activity *ma2:m2:1* at decision point *d1:m2:1*. Note that the time has not yet changed. We are still observing the construction of a new mission situation.



C



D

Figure 13: Mission Composition Transition C) An Effect Of Transition *Add Temporal System Requirements* D) An Effect Of Transition *Execute Mission Time Step*

Figure C) shows the effect of the transition *add temporal system requirements*. It just assigns the end of the time frame to mission activity ma2:mq:1. In Figure D) the transition *execute mission time step* that changes the mission status. The time is set to 9 and the token from *d2:m1:1* is moved to *mp2:m2:1*.

Now we can imagine the colour of the system requirements scenario rsp. the data type of the mission situation. It is the set of extended causal nets that can build upon the mission activity instances and the decision point instances. The transition *add functional requirements* just build the union of two extended causal nets to build a new extended causal net. The transition *add temporal system requirements* changes time attributes of the current extended causal net and the *execute mission time step* changes the state attributes of the current extended causal net. For details on the formalization see [Mackenthun04].

The execution sequence of the ARM is a simple cycle. During each time step the following sequence is fired: *generate use scenario*, *identify missions*, *accept missions*, *plan mission*, *add functional system requirements*,*schedule mission*, *timing mission*, *add temporal system requirements*, *execute mission time step*, *report mission situation*.

The success of the ARM will depend on two factors that we have stated before. At first the requirements engineer must be able to formulate the requirements specification as mentioned above. Secondly, the designer must be able to understand the requirements scenarios that are produced by the machine.

We see no problems in defining sets and data type or to understand produced requirements scenarios. To convince the reader that also the formulation of the mission definition transition is performable we show the definition of a transition *plan mission* in the next section for an example.

7 A Small Example: An Airbag Control System

We present a small example for a fictive airbag control calculation that is specified by the integrator of a car. An airbag control solution might consist of many software solution components. We assume that the integrator is interested in the interaction of solution components that are developed by different suppliers. So he has to specify the requirements in such a way that the interaction of software of different suppliers can be checked. Do not confuse the system requirements scenarios with the run of a solution, even though they look similar. Solutions are specified on a different abstraction levels. In this example we concentrate on explaining mission definition transition *plan mission* and the functional system requirements.

Figure 14 shows four principle system requirements scenarios form the integrator's view. The airbag calculation is performed in two activities *airbag1* and airbag2. For *airbag1* exists a solution from supplier *sp1* (named *airbag_sp1*) and from *sp3* and for *airbag2* exists solutions from supplier *sp2* and *sp3*. We assume that on the one hand the solutions form supplier *sp1* and *sp2*, and on the other hand the solutions from supplier *sp3* fit together. We have the following requirements scenarios. In A) is is assumed that the result of the first calculation is that no firing of the aribag is needed. In B) we assume that the situation after the first calculation is critical. The airbag needs to be fired. Therefore the second calculation has to take place. In C) and D) the first calculation has not met the deadline. An alternative calculation is required. In C) the alternative calculation ends in a normal, in D) in a critical situation. In the figure to each decision point the assumed decision condition is assigned (*airbag_normal* etc). They have to be evaluated on the execution situation.

From these scenarios we can extract the information for the coloured places in the machine: The mission are *M*={*airbag*} and the set of mission parts is *MP*={*(airbag,(b1,airbag1_sp1,pe1))*, *(airbag,(pe1,airbag2_sp2,e1))*, *(airbag,(b2,airbag1_sp3,pe2))*, *(airbag,(pe2,airbag2_sp2,e2))*}. This determines all colours of the grey result places.

Figure 15 shows the decision tree described as another coloured net. When a mission

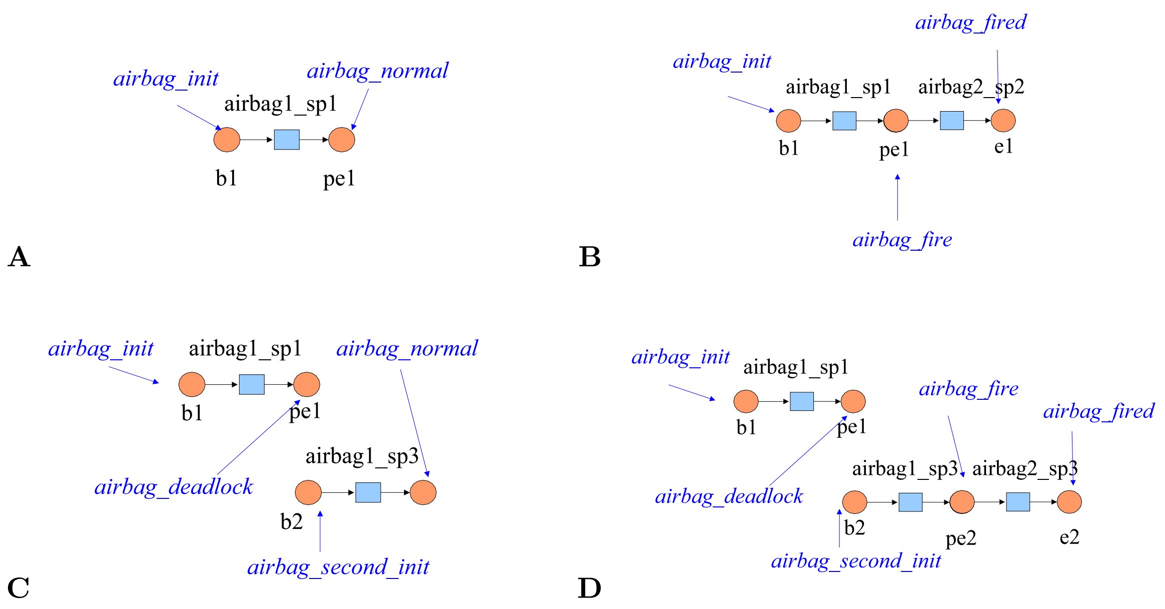


Figure 14: The airbag: A) calculated, not fired B) calculated, fired C) calculated, deadlocked, calculated, not fired D) calculated, deadlocked, calculated, fired

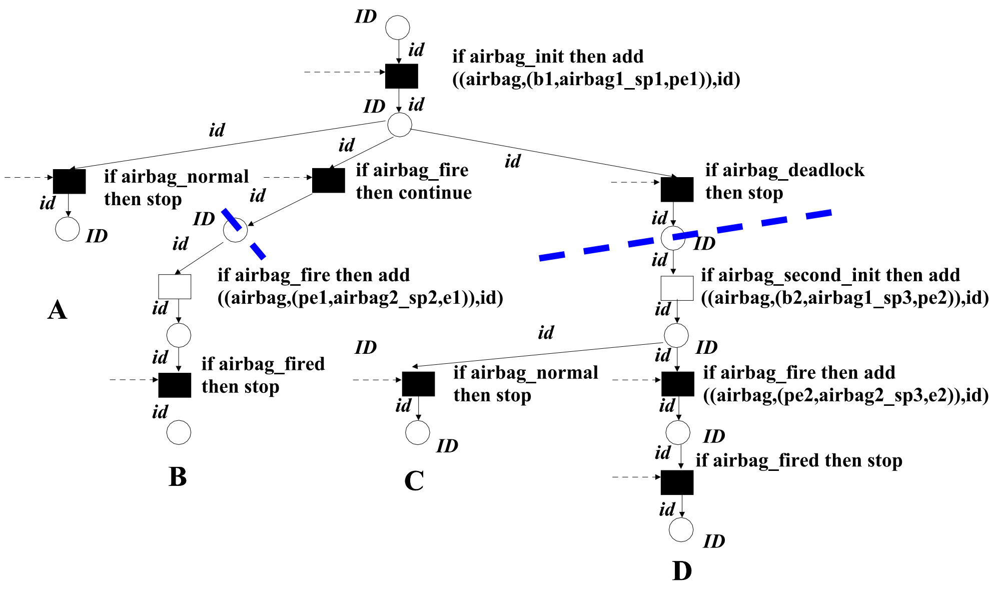
exists the identifier of that mission is lying on the starting place of the decision tree. When a time trigger occurs and the condition *airbag_init* holds, the first mission part is planned and send to the transition, who adds it to the functional requirements scenario. At the end of each path of the tree the respective scenario of figure 14 is mentioned. The black transitions wait for a trigger. The white transitions are fired immediately after the black, when their condition is true.

This decision tree as coloured net should just show one possibility of a description language for executable models of an algorithm for the *plan mission* transition that can be activated via time-triggering. We will not fix the description language in our approach.

In figure 16 we show an example of the work of the machine. Transition *plan mission* detects that condition *airbag_fire* is true and lets the machine send a respective token to transition *add functional system requirements* that changes the requirements.

8 Discussion, Conclusion and Future Work

The paper shows an approach to specify required constraints via an abstract requirements machine. With this machine functional and temporal requirements scenarios can be generated. The starting point for this approach was the lack of definition of system requirements scenarios. Some approaches define system requirements as rules. For RTCS these are to inflexible. Describing the required behaviour where the requirements can change dramatically because a deadline is missed, cannot be defined by rules because the complexity of the precondition of that rule would become an unmanageable complexity. So we decided to build a machine that generates the system requirements scenario depending on the current execution situations. This leads us to the formal specification approaches. The implicit definition of



if airbag_init then add ((airbag,(b1,airbag1_sp1,pe1)),id)
if airbag_normal then stop
if airbag_fire then continue
if airbag_deadlock then stop
if airbag_fire then add ((airbag,(pe1,airbag2_sp2,e1)),id)
if airbag_second_init then add ((airbag,(b2,airbag1_sp3,pe2)),id)
if airbag_fired then stop
if airbag_normal then stop
if airbag_fire then add ((airbag,(pe2,airbag2_sp3,e2)),id)
if airbag_fired then stop
ID
id
A
B
C
D

Figure 15: Decision Tree

partial ordered requirements scenarios by the formal specification approaches do not fulfil the needs for an easy formulation of requirements specification. Therefore we move to the explicit description of the requirements scenarios and their manipulation - the ARM.

The ARM is a coloured Petri net. We restricted the machine according to our methodological needs.

We define one of the places as the the location where system requirements scenarios are built. The token on that place is again a kind of Petri net: an extended causal net. This follows an idea in [Valk 91] who describes partial ordered task systems which are moving through a network of functional units. The network of functional units was a Coloured net, where the token where causal nets representing the task systems. By this, we meet the needs for easily understandable requirements scenarios.

The second restriction of the coloured Petri net is the definition of five mission definition transitions. This restriction is according to a natural way of defining missions: identification, acceptance, planning, scheduling, timing. This restriction allows the requirements engineer to specify the mission definitions rsp. the requirements scenarios definitions in a natural way.

The computer takes an assisting role, as for human beings, the generation of scenarios from different models in the way, the ARM works is hard work. The user of the requirements definition has to switch from one model semantics to another. This could leads to comprehension errors even though the requirements are specified in a mathematical precise way.

To conclude, we see the ARM as an important milestone on the way to dependable an secure RTCS. In detail, some of the benefits have already been discussed in section 4.

Our next steps are the application of the ARM in industrial projects and the description of the results of the integration activities that depends on the requirements specification: the requirements models for design and implementation and the constraint model for solution and service platform.

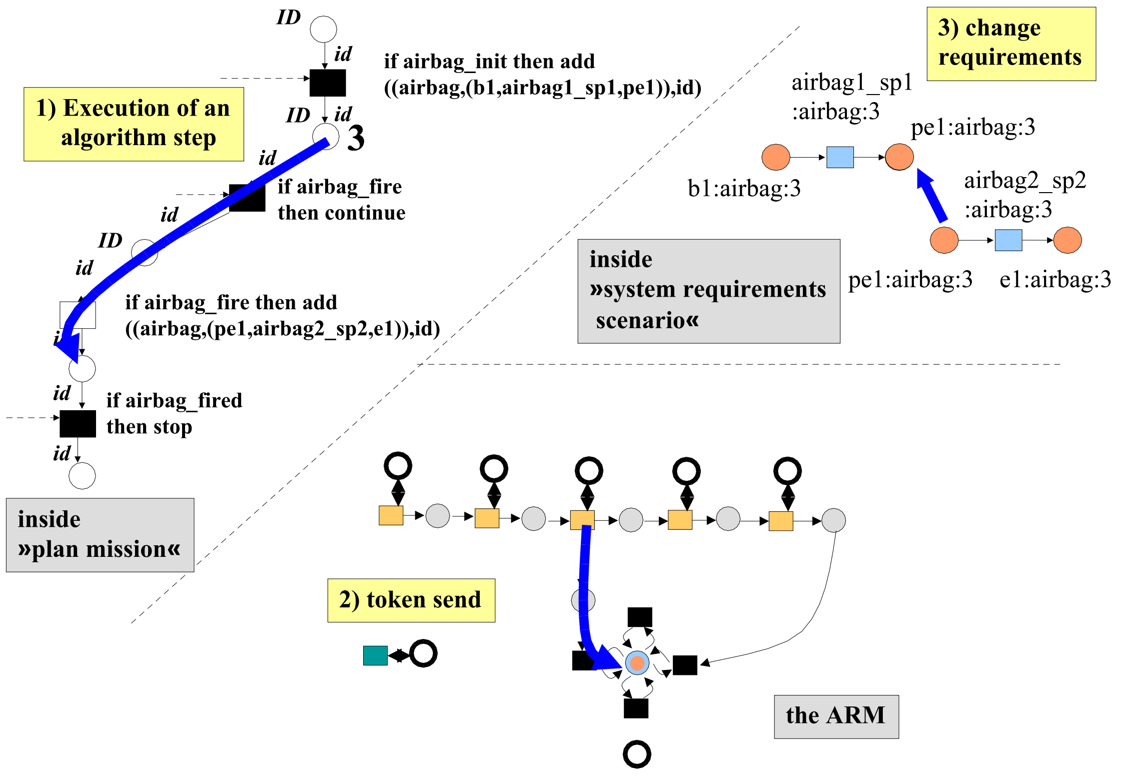

ID
id
if airbag_init then add ((airbag,(b1,airbag1_sp1,pe1)),id)
1) Execution of an algorithm step
ID
id
3
id
if airbag_fire then continue
id
ID
id
if airbag_fire then add ((airbag,(pe1,airbag2_sp2,e1)),id)
id
if airbag_fired then stop
id
inside »plan mission«
3) change requirements
airbag1_sp1 :airbag:3
pe1:airbag:3
b1:airbag:3
airbag2_sp2 :airbag:3
inside »system requirements scenario«
pe1:airbag:3
e1:airbag:3
2) token send
the ARM

Figure 16: Example of machine work

Beside the requirements specification, Fraunhofer ISST is working on the model basing of the development and the integration processes and their connection via model repositories and transformations. In addition, the model basing of the domain engineering and the product line engineering belong to our research portfolio. We have already achieved a lot of results in these areas (e.g. [Billig 03] [Borusan et. al. 04],[Große-Rhode, Mann 04a],[Große-Rhode, Mann 04b],[Hardt, Feldo 04],[Hardt et. al. 04]) and still see an increasing number of challenges for applied researcher in the future.

References

[**Bellini et. al. 00**]: P. Bellini, R. Mattolini, P. Nesi: Temoral logics for real-time system specification, ACM Computing Surveys, ACM Press, New York, 2000.

[**Billig 03**]: A. Billig: A domain repository based on semantic web technology (in German), in R. Tolksdorf and R. Eckstein (eds.), Berliner XML Tage 2003, 13 - 15 October 2003, Berlin, XML-Clearinghouse, 2003.

[**Borusan et. al. 04**] A. Borusan, M. Große-Rhode, R. Mackenthun and members of the dependable systems department: Model based system development, Internal reports on an industrial project with the BMW Group, Fraunhofer ISST, Berlin, Germany, 2001-2004.

[**Jensen 97**] K. Jensen: Coloured Petri nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1-3, Monographs in Theoretical Computer Science, Springer-Verlag, 1997.

[**Große-Rhode, Mann 04a**] M. Große-Rhode, S. Mann: Model-based Systems Engineering in the Automobile Industry, position paper sent to Positions and Experiences International workshop on solutions for Automotive Software Architectures: open standards, reference architectures, product line architectures, and architecture definition languages", Boston, Massachusetts, USA, August 31, 2004.

[**Große-Rhode, Mann 04b**] M. Große-Rhode, S. Mann: Model-based Development and Integration of Embedded Components: an Experience Report from an Industry Project, In: Proc. Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA), Satellite event of ETAPS 2004, 27 March - 4 April 2004, Barcelona, Catalonia, Spain, Pages 112-117.

[**Hardt et. al. 04**] M. Hardt, R. Mackenthun, J. Bielefeld: Integrating ECUs in Vehicles - Requirements Engineering in Series Development. In RE, procedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE 2002), 9-13 September 2002, Essen, Germany, IEEE Computer Society, 2002, Pages 227-236.

[**Hardt, Feldo 04**] M. Hardt, M. Feldo: The AUTOSAR specifications, Internal notes on the ongoing work in the AUTOSAR development partnerships, Fraunhofer ISST, Berlin, Germany, 2003-2004.

[**Heitmeyer, Mandrioli 95**] C. Heitmeyer, D. Mandrioli: Formal methods for real time computing, John Wiley & Sons Ltd, 1995.

[**Mackenthun 04**] R. Mackenthun, Unifying The Integration Of Real Time Computing Systems - An Abstract Machine To Specify Required Constraints, PhD thesis (to appear in 2004).

[**Maier, Moldt 01**] C. Maier, Daniel Moldt: Object Coloured Petri Nets - a Formal Technique for Object Oriented Modelling. In: G. Agha, F. de Cindio, and G. Rozenberg (eds.), Concurrent Object-Oriented Programming and Petri Nets, Nummer 2001 in Lecture Notes in Computer Science, Pages 402-424. Springer-Verlag, Berlin, 2001.

[**Popper 34**] Karl R. Popper, The Logic of Scientific Discovery, 1934.

[**Valk 91**] R. Valk: Modelling Concurrency by Task/Flow EN Systems Proceedings 3rd Workshop on Concurrency and Compositionality, GMD-Studien Nr. 19, Gesellschaft f. Mathematik und Datenverarbeitung, St. Augustin, Bonn, 1991.

[**Valk 98**] R. Valk: Petri Nets as Token Objects: An Introduction to Elementary Object Nets In: Jörg Desel, Manuel Silva (eds.): Application and Theory of Petri Nets; Lecture Notes in Computer Science, Vol. 1420, Springer-Verlag, Berlin, June 1998, Pages 1-25.

On 1-soundness and Soundness of Workflow Nets[*]

Lu Ping[1]   Hu Hao[2]   Lü Jian[2]

[1](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China; Email: luping@ics.nju.edu.cn; Phone: +86 25 83593694)

[2](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

Abstract: Soundness is the key property for evaluating “good” workflow nets (WF-nets). In this paper, we examine several kinds of WF-nets with well-known properties and prove that for these WF-nets, 1-soundness implies soundness. For Free-choice WF-nets and Well-handled WF-nets, 1-soundness, hence soundness, can be solved in polynomial time. Based on these results, we introduced a special kind of WF-nets – WRI WF-nets which is inherently sound and powerful enough for many modeling problems.

Keywords: Workflow nets; 1-soundness; Soundness; WRI WF-nets

Introduction

Petri nets are frequently used to model and analyze workflows. In [1], W.M.P van der Aalst used workflow nets (WF-nets) to model workflow processes. In [2], Kees van Hee et al. proposed the notion of *k*-soundness and soundness for WF-nets to judge whether a workflow model would behave correctly when executed. A WF-net is *k*-sound if any marking reached from *k* tokens in the initial place can reach the same *k*-tokens in the final place. Soundness is the property of *k*-soundness for every natural number *k*.

Soundness property is important for workflow models and could be preserved during workflow composition, which is a key mechanism in workflow modeling. It also allows us to analyze the concurrent execution of several workflow instances without introducing id-tokens. In [3], Kees van Hee et al. proved that soundness of WF-nets is decidable. A decision procedure was proposed to verify the soundness of arbitrary WF-nets. However, it is still to be investigated how to solve the problem of soundness efficiently and what complexity the algorithm would have.

In this paper, we try to tackle the problem of soundness for some kinds of WF-nets in another way, by establishing the relationship between 1-soundness and soundness. For these kinds of WF-nets, we prove that 1-soundness implies soundness. For Free-choice WF-nets and Well-handled WF-nets, the soundness problem can be decided in polynomial time.

In many cases, workflow models with balanced “split” and “merge” nodes and regular iterations are powerful enough for modeling problems. We introduce a special kind of WF-net – WRI WF-nets which have the above property and prove that they are inherently sound. WRI WF-nets support hierarchical workflow modeling naturally and ensure the soundness without the need of verification.

[*] Supported by NNSFC (60273034, 60233010), 863 Program of China (2002AA116010, 2001AA113090), 973 Program of China (2002CB312002), JSFC (BK2002203, BK2002409)

The remainder of the paper is organized as follows: the next section presents the basic concepts and notations of Petri nets. In Section 3, WF-nets and their soundness are defined; basic properties about soundness and composition of WF-nets are described. Section 4 establishes the relationship between 1-soundness and soundness for some kinds of WF-nets with well-known properties. In Section 5, we introduce WRI WF-nets and prove that they are inherently sound. The rationality of using WRI WF-nets to model workflow is described. Section 6 concludes the paper.

2 Basic Definitions and Notations of Petri nets

Definition 2.1 (Petri net).

(1) A Petri net is a 3-tuple $PN = (P, T, F)$ where

(a) P and T are finite and disjoint sets

(b) $F \subseteq (P \times T) \cup (T \times P)$

The elements of P are called places and the elements of T transitions.

(2) The preset of a node $x \in P \cup T$ is defined as $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$

The postset of $x \in P \cup T$ is defined as $x\bullet = \{y \in P \cup T \mid (x, y) \in F\}$

The preset (resp. postset) of a set is the union of the presets (resp. postsets) of is elements.

(3) The state (marking) of a Petri net is the distribution of tokens over places. Let M be a marking of net PN, p is one of its places. $M(p)$ is the number of token in p at state M. For two markings of PN: M_1 and M_2, $M_1 < M_2$ iff $\forall p \in P$, $M_1(p) < M_2(p)$. For $M_2 < M_1$, the marking $M = M_1 - (+) M_2$ iff $\forall p \in P$, $M(p) = M_1(p) - (+) M_2(p)$. Let p be a place and M be a marking, $M|p$ is such a marking that $M|p(p) = M(p)$ and $M|(q) = 0$ for $q \neq p$.

The number of tokens may change during the execution of the net. Transitions are the active components in a Petri nets: they change the state of the net according to the following firing rule:

(1) A transition t is said to be enabled iff each input place p of t contains at least one token.

(2) An enabled transition may fire. If transition t fires, t consumes one token from each input place p of t and produces one token for each output place p of t.

Given a Petri net $PN = (P, T, F)$ and a state M_0, we get a Petri net system (PN, M_0) (We also call (PN, M_0) a Petri net for simplicity). We have the following notations:

(1) $M_0 \xrightarrow{t} M_1$: transition t is enabled in state M_0 and firing t in M_0 results in state M_1.

(2) A firing sequence is described as $\sigma = t_1 t_2 \ldots\ldots t_n$, if $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$.

A state M_n is called reachable from M_0 (notation $M_0 \xrightarrow{*} M_n$) iff there exists a firing sequence $\sigma = t_1\ t_2 \ldots t_n$ so that $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} M_n$. We use $[M_0>$ to represent the set of all markings reachable from M_0. For a place p, we use $[p]$ ($[p^k]$) to represent the marking that place p has only one token (k tokens) and no other place in a net has any token.

Definition 2.2 (Liveness). A Petri net system (PN, M) is live iff for every reachable state $M^{'}$ and every transition t there is a state $M^{''}$ reachable from $M^{'}$, which enables t.

Definition 2.3 (Boundness, Safeness). A Petri net (PN, M) is bounded iff for every place p there exists a natural number bound n so that for each state $M^{'} \in [M>$, $M^{'}(p) \leq n$. The net is safe iff for each place the bound is 1.

Definition 2.4 (Path, Elementary). Let $PN = (P, T, F)$ be a Petri net. A path C from a node n_1 to a node n_k is a sequence $<n_1\ n_2 \ldots\ldots n_k>$ such that $<n_i, n_{i+1}> \in F$ for $1 \leq i \leq k-1$. C is elementary iff, for any two nodes n_i and n_j on C, $i \neq j \Rightarrow n_i \neq n_j$. Let $\alpha(C) = \{n_1, n_2, \ldots, n_k\}$. C_1C_2 represents the concatenation of paths C_1 and C_2 if the last node of C_1 and the first of C_2 are identical.

Definition 2.5 (Siphon, Trap, Marked trap, Siphon-trap property). Let $PN = (P, T, F)$ be a Petri net. A non-empty set $H \subseteq P$ is called a siphon (trap) iff $\bullet H \subseteq H \bullet$ ($H \bullet \subseteq \bullet H$). Let H be a siphon (trap), H is called minimal iff there is no siphon (trap) included in H as a proper subset. A trap is marked iff at least one place in the trap is marked. A Petri net system (PN, M) holds siphon-trap property if every siphon of PN contains a marked trap at M.

Definition 2.6 (Well-formedness). A net PN is well-formed if there exists a marking M_0 of PN such that (PN, M_0) is a live and bounded system.

Definition 2.7 (Free-choice, Extended Free-choice, Asymmetric choice). Let $PN = (P, T, F)$ be a Petri net. PN is a free-choice net iff $\forall p \in P \quad |p\bullet| \leq 1$ or $\bullet(p\bullet) = \{p\}$; PN is an extended free-choice net iff $\forall p_1, p_2 \in P \quad p_1\bullet \cap p_2\bullet \neq \emptyset \Rightarrow p_1\bullet = p_2\bullet$; PN is an asymmetric choice net iff $\forall p_1, p_2 \in P \quad p_1\bullet \cap p_2\bullet \neq \emptyset \Rightarrow p_1\bullet \subseteq p_2\bullet$ or $p_1\bullet \supseteq p_2\bullet$.

Definition 2.8 (State machine, S-component, S-coverable). Let $PN = (P, T, F)$ be a Petri net. PN is a state machine iff $\forall t \in T \quad |t\bullet| = 1$ and $|\bullet t| = 1$. A subnet $PN_s = (P_s, T_s, F_s)$ is called an S-component of PN if $P_s \subseteq P$, $T_s \subseteq T$, $F_s \subseteq F$, PN_s is strongly connected, PN is a state machine, and for every $q \in P_s$ and $t \in T$: $(q, t) \in F \Rightarrow (q, t) \in F_s$ and $(t, q) \in F \Rightarrow (t, q) \in F_s$. PN is S-coverable iff for any node (transitions and places) there exists an S-component which contains this node.

3 Workflow Nets, Its Soundness and Composition

3.1 Workflow Nets, Its 1-soundness, *K*-soundness and Soundness

Definition 3.1 (WF-net [1]). A Petri net $PN = (P, T, F)$ is a WF-net (Workflow net) iff:

(1) PN has two special places: i and o. Place i is a source place: $\bullet i = \emptyset$. Place o is a sink place: $o\bullet = \emptyset$.

(2) If we add a transition t^* to PN so that $\bullet t^* = \{o\}$ and $t^*\bullet = \{i\}$, then the resulting Petri net is strongly connected.

Definition 3.2 (*K*-soundness [12]). A WF-net $PN = (P, T, F)$ is k-sound for a natural number k if and only if:

(1) $\forall M\ ([i^k] \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} [o^k])$
(2) $\forall M\ ([i^k] \xrightarrow{*} M \ \wedge\ M \geq [o^k]) \Rightarrow (M = [o^k])$
(3) $\forall t \in T\ \exists M, M'\ [i^k] \xrightarrow{*} M \xrightarrow{t} M'$

Remark: In Definition 3.2, the second requirement could be omitted since requirement (1) implies it. If the requirement (1) is met and there is a reachable marking $M > [o^k]$, then from M, the state $[o^k]$ can be reached. But this is impossible since o is a sink place. We just retain it here to let our definition be conformed to the original definition made in [1] and [12].

Definition 3.3 (Soundness). A WF-net *PN* is sound iff it is *k*-sound for every natural number *k*.

Definition 3.4 (Extended WF-net [1]). Given a workflow net $PN = (P, T, F)$, the extended Petri net of *PN*, represented by $PN^* = (P^*, T^*, F^*)$, is defined as follows:
(1) $P^* = P$
(2) $T^* = T \cup \{t^*\}$
(3) $F^* = F \cup \{(o, t^*), (t^*, i)\}$

Proposition 3.1 ([1]). A WF-net *PN* is 1-sound iff $(PN^*, [i])$ is live and bounded.

Since when we use WF-nets in workflow modeling, the 1-soundness property is the basic property WF-nets should have, it is reasonable to examine the *k*-soundness ($k > 2$) and soundness of WF-nets under the presumption that they are 1-sound. Now we give some intuitive results toward *k*-soundness of WF-nets. In next section, we would investigate the soundness property under the presumption of 1-soundness for several kinds of WF-nets.

Proposition 3.2 If a 1-sound WF-net *PN* is *k*-sound, then for all natural numbers $p < k$, *PN* is *p*-sound.
Proof. Suppose there is a natural number $p < k$ so that *PN* is not *p*-sound. Since *PN* is 1-sound, requirement (3) for *q*-soundness ($q > 1$) must be met. Then the first requirement for *p*-soundness in Definition 3.2 must not hold for *PN*. For $(PN, [i])$, there would be a firing sequence σ_1 so that $[i] \xrightarrow{\sigma_1} [o]$. For $(PN, [i^p])$, there would be a firing sequence σ_2 so that $[i^p] \xrightarrow{\sigma_2} M$ and $[o^p]$ can not be reached from *M*. Now for $(PN, [i^k])$, we can fabricate such firing sequence that $\sigma = (\sigma_1)^{(k-p)}\sigma_2$. Let $[i^k] \xrightarrow{\sigma} (M+[o^{(k-p)}])$, since *o* is a sink place, from $(M+[o^{(k-p)}])$, the marking $[o^k]$ can not be reached. This contradicts with the fact that *PN* is *k*-sound. So *PN* must also be *p*-sound.

Proposition 3.3 If a 1-sound WF-net *PN* is not *k*-sound, then for all natural numbers $p > k$, *PN* is not *p*-sound.
Proof. Can be proved analogously as Proposition 3.2.

The above properties do not hold if the WF-net is not 1-sound. For example, in Figure 3.1, *PN* is 3-sound but neither 1-sound nor 2-sound.

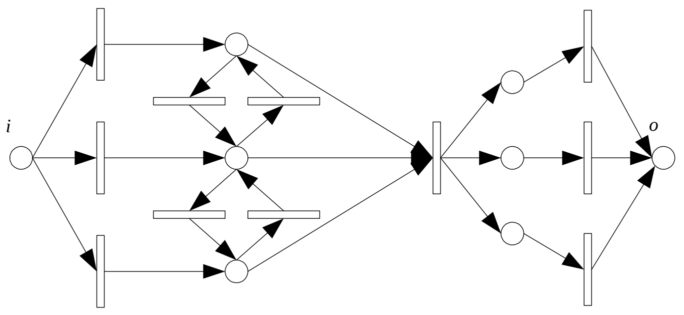


Figure 3.1 3-sound WF-net but neither 2-sound nor 1-sound

3.2 Composition of WF-nets

Composition is a key mechanism in workflow modeling. The *k*-soundness/soundness property is meaningful largely because 1-soundness is not compositional. In this section, we provide a way to realize composition of workflow for the convenience of introducing iteration to our WRI WF-nets later in this paper. This way of composition coincides exactly with the "transition refinement" introduced in [2].

Theorem 3.1. Let $PN_1 = (P_1, T_1, F_1)$, $PN_2 = (P_2, T_2, F_2)$ be two WF-nets such that the source and sink places in PN_2 are i_2 and o_2, $T_1 \cap T_2 = ø$, $P_1 \cap P_2 = ø$, $t_1 \in T_1$ and t_a, $t_b \notin T_1 \cup T_2$. $PN_3 = (P_3, T_3, F_3)$ is the composition of PN_1 and PN_2 obtained by replacing t_1 in PN_1 by PN_2 ($PN_3 = PN_1 \otimes t_1\, PN_2$) if:

$$P_3 = P_1 \cup P_2$$
$$T_3 = (T_1 \backslash \{ t_1\}) \cup T_2 \cup \{t_a, t_b\}$$
$$F_3 = \{(x, y) \mid (x, y) \in F_1 \wedge x \neq t_1 \wedge y \neq t_1\} \cup \{(x, y) \mid (x, y) \in F_2\} \cup$$
$$\{(x, t_a) \mid (x, t_1) \in F_1\} \cup \{(t_b, y) \mid (t_1, y) \in F_1\} \cup \{(t_a, i_2), (o_2, t_b)\}$$

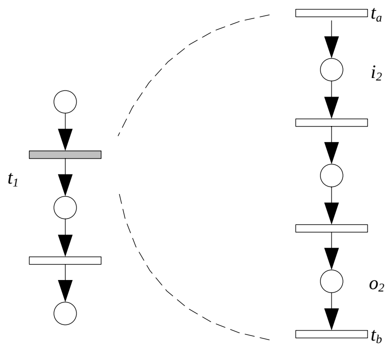


Figure 3.2 Composition of WF-nets

In [2], it is proved that soundness is compositional. Now we prove that when we compose a *k*-sound WF-net with a sound one, the *k*-soundness property is also maintained. This property is useful during workflow composition when we only want to ensure the 1-soundness of the result

WF-net.

Proposition 3.4 ([2]). Let PN_1, PN_2 be sound WF-nets and t be a transition of PN_1, WF-net $PN_3 = PN_1 \otimes t\ PN_2$ is also sound.

Theorem 3.2. Let PN_1 be k-sound WF-net, PN_2 be sound WF-net and t be a transition of PN_1. $PN_3 = PN_1 \otimes t\ PN_2$ is also k-sound.

Proof. By the proofs of Theorem 7, Lemma 8 and Theorem 9 of [2], we relabel all the transitions of PN_2 and t_b as τ, PN_1 and PN_3 are weakly WF-bisimilar [2]. Now suppose $[i^k] \xrightarrow{*} M$ within PN_3, there exists a state M' of PN_1 with M R M' (R is the WF-bisimulation relation describe in [2]) such that $[i^k] \xrightarrow{*} M'$ within PN_1. Since PN_1 is k-sound, we have $M' \xrightarrow{*} [o^k]$, then there exists a state M'' of PN_3 such that $M' \xrightarrow{*} M''$ and $M'' = [o^k]$ within PN_3. So requirement (1) of k-soundness holds for PN_3. In PN_3, for those transitions also in PN_1 and the transition t_a, they have the chance to fire since PN_3 and PN_1 are weakly WF-bisimilar and they can be fired in PN_1; for those transitions in PN_2 and t_b, they can be fired since PN_2 is sound and t_a can be fired. So requirement (3) also holds and PN_3 is k-sound.

4 Establishing Relationship with 1-soundness and Soundness

In [3], Kees van Hee et al. proved that soundness of workflow nets is decidable. Although their definition of soundness is a little different from ours (they omit the requirement (3) of Definition 3.2 but deals with the same criterion in BWF-nets [3]), using the decision procedure they provided, the soundness problem in our paper could also be solved. However, as stated in [3], for the soundness decision procedure, the authors focused on clarity rather than efficiency; it is still to be investigated how to solve the soundness in an efficient manner and what complexity the decision algorithm would have.

In this section, we try to solve the soundness problem for several kinds of WF-nets in a different manner. By proving that for these WF-nets, 1-soundness implies soundness, we only need to verify the 1-soundness to ensure soundness. For those WF-nets such as Free-choice WF-nets and Well-handled WF-nets whose 1-soundness could be solved effectively, so does the soundness problem for them.

4.1 Free-Choice, Extended Free-Choice and ST-Asymmetric Choice WF-nets

Free-choice nets, extended free-choice nets and asymmetric choice nets are three kinds of related Petri nets which have been studied extensively. The liveness of these nets has strong relation with siphons and traps. For free-choice and extended free-choice nets, efficient algorithms exist to determine their well-formedness. In [1], the author states that for almost all WFMS, only constructs that are comparable to free-choice WF-net are allowed. Therefore, it makes sense to study the soundness of workflow nets that are free-choice.

Definition 4.1 (FC WF-net, EFC WF-net). A WF-net PN is a FC WF-net (Free-Choice WF-net) if PN is a free choice net. A WF-net PN is an EFC WF-net (Extended Free-Choice

WF-net) if *PN* is an extended free choice net.

Before we prove that for FC WF-nets and EFC WF-nets, their 1-soundness implies soundness, we study a more general kind of WF-net – ST-AC WF-nets and prove for them, 1-soundness implies soundness.

Definition 4.2 (ST-AC WF-net). A WF-net *PN* is a ST-AC WF-net if PN^* is an asymmetric choice net and every siphon of it contains at least a trap.

Lemma 4.1. Let $PN = (P, T, F)$ be a 1-sound ST-AC WF-net, then for any natural number k, $(PN^*, [i^k])$ is live and bounded.

Proof. Since *PN* is a 1-sound ST-AC WF-net, $(PN^*, [i])$ is live and bounded and it is a ST-AC Petri net. Then every minimal siphon of PN^* is marked at $[i]$ (Theorem 5.1 [5]). Then at marking $[i^k]$, every siphon of PN^* must also be marked. So $(PN^*, [i^k])$ is also live and bounded (Theorem 5.1 [5]).

Theorem 4.1. Let $PN = (P, T, F)$ be a 1-sound ST-AC WF-net, *PN* is *k*-sound for any natural number *k*.

Proof. For an arbitrary natural number *k*, since *PN* is 1-sound, the third requirement of *k*-soundness in Definition 3.2 holds. We only need to prove requirement (1). Now suppose *PN* is a 1-sound ST-AC WF-net and for $(PN, [i^k])$, $\exists M$ reachable from $[i^k]$ so that from marking *M*, $[o^k]$ could not be reached. Let $M \xrightarrow{\sigma} M'$ for *PN* so that from M' no more transitions could put tokens in *i*. For $(PN^*, [i^k])$, *M* is also reachable from $[i^k]$, from marking *M*, we have $M \xrightarrow{\sigma} M'$ for PN^*. Since $(PN^*, [i^k])$ is live, $M'(o) > 0$. Let $M'' = M' - M'|o$, since for *PN* from M' no more tokens could be put into the sink place, for PN^*, for each marking reachable from M'', t^* would not be enabled. So (PN^*, M'') is not live but bounded. For PN^*, every minimal siphon is an S-component (Theorem 4.2 [5]) and since $(PN^*, [i^k])$ is live, every minimal siphon must contain place *i*. For $(PN^*, [i^k])$, every siphon has *k* tokens; at M', they would also has *k* tokens. Since $M'(o) < k$ (otherwise $(PN^*, [i^k])$ would not be bounded), at M'', every siphon of PN^* would be marked. According to Theorem 5.1 in [5], (PN^*, M'') is live which contradicts with our previous assertion. So we can conclude requirement (1) must also holds and *PN* is *k*-sound for any natural number *k*.

Corollary 4.1. For ST-AC WF-net, 1-soundness implies soundness.

Now we can conclude that for FC WF-nets and EFC WF-nets, 1-soundness implies soundness since if they are 1-sound, then they are also 1-sound ST-AC WF-nets.

Corollary 4.2. For EFC WF-net, 1-soundness implies soundness.

Proof. For 1-sound EFC WF-net *PN*, PN^* is live and bounded. Then every siphon of PN^* must contain a trap (Commoner’s Theorem [4]). So *PN* is also a ST-AC WF-net which has the property that 1-soundness implies soundness.

Corollary 4.3. For FC WF-net, 1-soundness implies soundness.
Proof. A FC WF-net is also an EFC WF-net, so its 1-soundness also implies soundness.

Proposition 4.1. The following problem could be solved in polynomial time:
Given an EFC WF-net, to decide if it is 1-sound.
Proof. Since the problem of deciding whether an extended free-choice system is live and bounded could be solved in polynomial time (Corollary 6.18 [4]), according to Proposition 3.1, the above problem could also been solved in polynomial time.

The following corollary shows that, for FC WF-nets and EFC WF-nets, there are efficient algorithms to decide soundness.

Corollary 4.4. The following problem could be solved in polynomial time:
Given an EFC WF-net or FC WF-net, to decide if it is sound.

4.2 Well-handled WF-nets and ENSeC WF-nets

The well-handled property is also a desired property of workflow net mentioned in [1]. From a modeler's view, such property enforces the balance of "split" and "merge" nodes which could help prevent ill behavior in workflow execution.

Definition 4.3 (Well-handledness [1]). A Petri net *PN* is well-handled iff, for any pair of nodes *x* and *y* such that one of the nodes is a place and the other a transition and for any pair of elementary paths *C1* and *C2* leading from *x* to *y*, $\alpha(C1) \cap \alpha(C2) = \{x, y\} \Rightarrow C1 = C2$.

Definition 4.4 (WH WF-net). A WF-net *PN* is a WH WF-net (Well-handled WF-net) iff PN^* is well-handled.

We also find that for a kind of WF-nets more general than WH WF-nets, their soundness could be implied by 1-soundness. Therefore, we also solve the problem that for WH WF-nets, 1-soundness implies soundness.

Definition 4.5 (Conflict-free, ENSeC net [6]). Let *PN* be a Petri net and $C = <n_1, \dots, n_k>$ be a path in *PN*, *C* is conflict-free iff for any transition n_i of the path, $j \neq i-1 \Rightarrow n_j \notin \bullet n_i$. Let *PN* be a Petri net, *PN* is an Extended Non-Self Controlling (ENSeC) net iff for every pair of transitions t_1 and t_2 such that $\bullet t_1 \cap \bullet t_2 \neq \emptyset$, there does not exist a conflict-free path leading from t_1 to t_2.

Definition 4.6 (ENSeC WF-net). A WF-net *PN* is an ENSeC WF-net iff PN^* is a ENSeC net.

Proposition 4.2 ([6]). Let (PN, M_0) be a marked ENSeC net. If (PN, M_0) is live and bounded, *PN* is S-coverable. And if (PN, M_0) is bounded, the four following properties are equivalent:
(1) (PN, M_0) is live

(2) Every minimal siphon is a trap and no siphon is empty in M_0
(3) Every minimal siphon is a marked state-machine in M_0
(4) The siphon-trap property holds for (PN, M_0)

Lemma 4.2. Let $PN = (P, T, F)$ be a 1-sound ENSeC WF-net, then for any natural number k, $(PN^*, [i^k])$ is live and bounded.
Proof. Since PN is 1-sound, by Proposition 4.2, PN^* is S-coverable. Then it is obvious that $(PN^*, [i^k])$ is bounded since in any S-component of PN, the token number would not change. Also we have that the siphon-trap property holds for $(PN^*, [i^k])$ since it must hold for $(PN^*, [i])$. By Proposition 4.2, $(PN^*, [i^k])$ must also live. So we get that $(PN^*, [i^k])$ is live and bounded.

Theorem 4.2. Let $PN = (P, T, F)$ be a 1-sound ENSeC WF-net, PN is k-sound for any natural number k.
Proof. PN is 1-sound, so the requirement (3) of k-soundness holds. We only need to prove the requirement (1) of k-soundness. Now suppose PN is a 1-sound ENSeC WF-net and for $(PN, [i^k])$, $\exists M$ reachable from $[i^k]$ so that from marking M, $[o^k]$ could not be reached. For PN, let $M \xrightarrow{\sigma} M'$ so that from M' no more transitions could put tokens into i. For $(PN^*, [i^k])$, M is also reachable from $[i^k]$, from marking M, we have $M \xrightarrow{\sigma} M'$. Since $(PN^*, [i^k])$ is live, $M'(o) > 0$. Let $M'' = M' - M'|o$, since in PN from M' no more tokens could be put into the sink place, for PN^*, no marking reachable from M'' would enable t^*. So (PN^*, M'') is not live but bounded. Since every minimal siphon in PN^* is a state-machine and must contain place i, which implies that at M', every siphon would also have k tokens. Because $M'(o) < k$ (otherwise $(PN^*, [i^k])$ would not be bounded), at M'', every minimal siphon would also be marked. By Proposition 3.2, we have that (PN^*, M'') is live which contradicts with the previous assertion that (PN^*, M'') is not live. Now we can conclude that requirement (1) of k-soundness also holds and PN is k-sound.

Corollary 4.5. For ENSeC WF-net, 1-soundness implies soundness.

Corollary 4.6. For WH WF-net, 1-soundness implies soundness.
Proof. For a WH WF-net PN, we prove that it is also an ENSeC WF-net. Suppose that PN is not an ENSeC WF-net, then for PN^* there must be two transitions t_1 and t_2 in conflict so that there exists a conflict-free path C leading from t_1 to t_2. Let $p \in \bullet t_1 \cap \bullet t_2$, then p must not appear in C since otherwise C would not be conflict-free. Then from p to t_2, there exist two different paths $C_1 = <p, t_2>$ and $C_2 = <p, t_2>C$ so that $\alpha(C1) \cap \alpha(C2) = \{p, t_2\}$ and PN^* must not be well-handled which contradicts with the fact that PN is WH WF-net. Now the corollary follows immediately from the fact that for ENSeC WF-net, 1-soundness implies soundness.

Proposition 4.3 [1]. The following problem could be solved in polynomial time:
Give a WH WF-net, to decide if it is 1-sound.

The following corollary shows that, for WH WF-nets, there are efficient algorithms to decide soundness.

Corollary 4.7. The following problem could be solved in polynomial time:
Given a WH WF-net, to decide if it is sound.

5 WRI WF-nets

In the previous section, we examined several kinds of WF-nets whose soundness can be implied by 1-soundness. Upon some of them, deciding soundness could even be solved in polynomial time. Using them in workflow modeling is promising. However, for many business processes encountered in practice, these restricted workflow models are still too complex. We found in many cases, workflow models with balanced "split" and "merge" node and regular iterations are powerful enough for modeling problems. In this section, we introduce a kind of such WF-nets and prove that they are inherently sound. Thus, we can use them in workflow modeling without the verification of soundness.

5.1 WA WF-nets

WA WF-nets are WF-net that are well-handled and acyclic (cycle-free). While they could not model iterations, they make of the fundamental building blocks of WRI WF-nets described later.

Definition 5.1 (WA WF-net). A WF-net is a WA WF-net iff it is well-handled and acyclic

Proposition 5.1. For a WA WF-net *PN*, PN^* is free-choice.
Proof. We first prove that *PN* is free-choice. Suppose *PN* is not free-choice, then there must be two transition t_1 and t_2 such that $\bullet t_1 \cap \bullet t_2 \neq \emptyset$ and $\bullet t_1 \setminus \bullet t_2 \neq \emptyset$. But this will violate the well-handled property of *PN* (see Figure 5.1). Since *PN* is free-choice, PN^* is also free-choice.

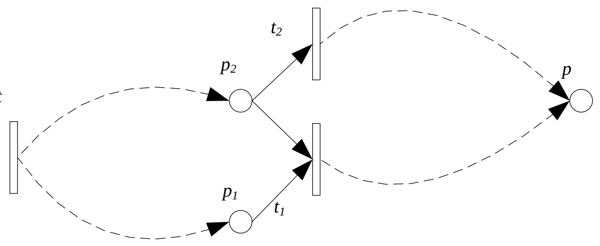


Figure 5.1 Violation of well-handled property: $<t \ldots p_1, t_2 \ldots p>$ and $<t \ldots p_2, t_1 \ldots p>$

Proposition 5.2. For a WA WF-net *PN*, PN^* is well-handled.
Proof. Suppose PN^* is not well-handled, then there are two elementary paths C_1, C_2, a place *p* and a transition *t* in PN^* so that $\alpha(C_1) \cap \alpha(C_2) = \{p, t\}$. Let C_1 and C_2 lead from *p* to *t*. It is easy to see that one of these two paths would contain the nodes *i*, *o* and t^* but the other contains neither of them since *PN* is well-handled. See Figure 5.2, C_1 is the path which does not contain *i*, *o* and t^*. $C_2 = C_{21} t^* C_{22}$ is the other path. From *t* we make a path C_x in *PN* towards place *o*, let the node p' be the first node in C_x that is also in C_{21}; p' must be a place since *p* is a place. Let the subpath of C_x from *t* to p' be C_3; the subpath of C_{21} from *p* to p' be C_5. Similarly, we lead a path

C_y in *PN* from *i* to *p* and let t' be the last node in C_y that is also in C_{22}; t' is a transition since *t* is a transition. Let the subpath of C_y from t' to *p* be C_4, the subpath of C_{22} from t' to *t* be C_6. Since *PN* is acyclic, there is no identical node in C_3 and C_4. Now we get two paths $C_a = C_4C_5$ and $C_b = C_6C_3$ so that $\alpha(C_a) \cap \alpha(C_b) = \{p', t'\}$ which contradicts with the fact that *PN* is well-handled. In case that C_1, C_2 are from *t* to *p*, such contradiction could be gotten analogously. So PN^* must also be well-handled.

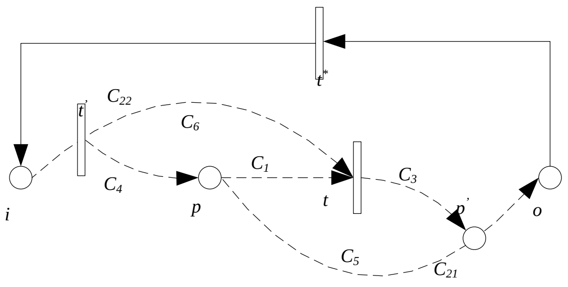


Figure 5.2 Violation of well-handled property: $C_6\ C_3$ and C_4C_5

Lemma 5.1. If *PN* is a WA WF-net, then $(PN^*, [i])$ is safe
Proof. Since PN^* is well-handled, it's easy to see that no circuit of PN^* has a PT- or TP-handle [7]. Then $(PN^*, [i])$ is bounded and covered by S-components [7]. We could know $(PN^*, [i])$ is safe since every S-component of it must have the place *i* (otherwise the S-component would not be strongly connected) and at the initial marking, only place *i* has one token.

Lemma 5.2. Let *PN* = (P, T, F) be a WA WF-net, then $(PN^*, [i])$ is live.
Proof. From Proposition 5.1 we know that $PN^* = (P^*, T^*, F^*)$ is free-choice. Now we prove that every minimal siphon of PN^* is a trap. Let *H* be an arbitrary minimal siphon of PN^*, let the subnet $PH = (H, \bullet H, \bullet F^*)$ in which $\bullet F^* = F^* \cap ((H \times \bullet H) \cup (\bullet H \times H))$, *PH* is strongly connected [13]. Then it's easy to see that *PH* contains place *i* and *o*. Now suppose *H* is not a trap, there must be a transition $t \in H\bullet$ such that $t\bullet \cap H = ø$. In *PN* there must be a path *C* from *t* to *o*, and let *x* be the first node in *C* that is contained in $H \cup \bullet H$. Such node *x* must exist for place *o* is in the path and is also contained in $H \cup \bullet H$. The node *x* could not be a place since all transitions in $\bullet x$ must also in $H \cup \bullet H$ and *x* would then not be the first node of path *C* in $H \cup \bullet H$. Thus, *x* is a transition, let the subpath of *C* from *t* to *x* be C_x. Let $p \in \bullet t$ and $p \in H$, since *PH* is strongly connected, let the elementary path in *PH* from *p* to *x* be C_2. Now we got two elementary paths $C_1 = \langle p, t \rangle C_x$ and C_2 so that $\alpha(C_1) \cap \alpha(C_2) = \{p, x\}$ but *p* is a place, *x* is a transition. This would contradict with the fact that PN^* is well-handled (Proposition 5.2). So there must be no such $t \in H\bullet$ and we have $H\bullet = \bullet H$ which implies that every minimal siphon in PN^* is a marked trap at [*i*]. From Commoner's Theorem, $(PN^*, [i])$ is live.

Theorem 5.1. If *PN* is a WA WF-net, then *PN* is 1-sound
Proof. Follows immediately from Lemma 5.1 and Lemma 5.2.

Corollary 5.1. If *PN* is a WA WF-net, then *PN* is sound.

Proof. Follows immediately from Theorem 5.1, Proposition 5.1 and Corollary 4.3.

5.2 WRI WF-nets

Since WA WF-nets are acyclic, they could not model iteration, which is one of the most important routing structures in workflow process. WRI (Well-handled with Regular Iterations) nets add regular iterations to WA WF-nets. In many cases, they are capable of modeling workflow process effectively.

Definition 5.2.

(1) A WA WF-net is a WRI WF-net.

(2) Let $PN_1 = (P_1, T_1, F_1)$, $PN_2 = (P_2, T_2, F_2)$ be two WRI WF-nets and $t \in T_1$. $PN_3 = PN_1 \otimes_t PN_2$ is a WRI WF-net.

(3) Let $PN_1 = (P_1, T_1, F_1)$, $PN_2 = (P_2, T_2, F_2)$ be two WRI WF-nets and $t \in T_1$. $PN_3 = PN_1 \otimes_t PN_2^*$ is a WRI WF-net.

(4) WRI WF-nets could only be obtained by (1), (2) and (3).

Remark: In Definition 5.2(3), the PN_2^* is not a WF-net. But we realize the composition with the same way we use in Theorem 3.1, that is, we add t_a and t_b as the pre-transition and post-transition of place *i* and *o* in PN_2^* and replace t_1 by the new net.

From the definition above, we can see that the iterations in WRI WF-nets are regular iterations, that is, iterations formed by a set of nodes with an entrance node and an exit node. The subnet formed by this set of nodes is strongly connected; the entrance node is the only node in the set that its preset contains nodes which is not in the set; the exit node is the only node in the set that its postset contains nodes which is not in the set. Iterations that are not regular are not allowed in WRI WF-nets.

Now we prove that for WRI WF-nets, they are inherently sound.

Theorem 5.2. WRI WF-nets are 1-sound

Proof. Suppose $PN_1 = (P_1, T_1, F_1)$, $PN_2 = (P_2, T_2, F_2)$ be two WRI WF-nets. $PN_3 = (P_3, T_3, F_3)$ is the WRI WF-net obtained by replacing t_1 in PN_1 by PN_2 or PN_2^*. It is easy to see that if $(PN_1^*, [i])$ and $(PN_2^*, [i])$ are live and safe, then $(PN_3^*, [i])$ is live and safe. According to our definition of WRI WF-nets, for each WRI PN_x, PN_x must be live and safe since the extended net of each WA WF-net is live and safe. According to Proposition 3.1, every WRI WF-net is 1-sound.

Proposition 5.3. For a WRI WF-net *PN*, PN^* free-choice.

Corollary 5.2. If *PN* is a WRI WF-net, then *PN* is sound.

Proof. Follows immediately from Theorem 5.2, Proposition 5.2 and Corollary 4.3.

5.3 Use WRI WF-nets to Model Workflow Processes

The most obvious restrictions upon WRI WF-nets are: (1) the “split” and “merge” nodes must be balanced; (2) only regular iterations are allowed. However, these two restrictions are reasonable and also appear in many other workflow models. In [8], two structural conflicts in workflow process model are identified: deadlock and lack of synchronization. Join OR-SPLIT paths with an AND-MERGE results into a deadlock conflict. Similarly, joining AND-SPLIT concurrent paths with an OR-MERGE results into unintentional multiple activation of nodes, which is called “lack of synchronization”. The well-handled property of WF-nets can balance AND/OR-SPLITS and AND/OR-MERGES to avoid “deadlock” and “lack of synchronization”. The main reason why we provide regular iterations as the only way to model loop control is somewhat similar to the reason why ‘goto’ is forbidden in many programming languages. Using only regular iterations ensures that the activities within the iteration part do not activate some other objects outside the iteration path while the iteration is in progress. Otherwise, such an activation would result in unintentional multiple execution of a workflow path that is not part of the iteration and to analyze the dynamic behavior of a workflow model would become difficult. In [9], iterations in workflow model that are not regular are regarded as syntactical errors. In CPMS system [11], our workflow model also could only depict regular iterations and software companies using our system to model their software processes did not complain about this restriction.

Since WRI WF-nets are capable of modeling sequential, parallel and iteration routings and are inherently sound, it is promising to model workflow process using WRI WF-nets. The recursive definition of WRI WF-nets suggests that WRI WF-nets supports hierarchical modeling: (1) First, the sketch of a workflow process is modeled by a WA WF-net; those iterations and subnets to be refined are represented by special transitions in the initial net. (2) Then, these special transitions are replaced by WA WF-nets or by WA WF-nets’ extended nets in which special transitions may also exist to represent the subnets or iterations to be modeled next. We will continue this building process as long as there are still special transitions in our WF-net. After all these special transitions are replaced in this way, we get a sound workflow model depicted by a WRI WF-net. An example of this building process is described by the Figure 5.3 (special transitions are shaded).

The example first appeared in [1]. We use this example here to illustrate the power of WRI WF-nets in workflow modeling. The workflow depicts the processing of complains. First the complaint is registered (task *register*), then in parallel a questionnaire is sent to the complainant (task *send_questionnaire*) and the complaint is evaluated (task *evaluate*). If the complainant is not returned within two weeks, the result of questionnaire is discarded (task *time_out*). Based on the result of evaluation, the complaint is processed or not. The actual processing of the complaint (task *process_complaint*) is performed until the result is checked (task *check_processing*) and passed. Here we use WRI WF-nets along with the stepwise modeling method described above to construct this workflow process. At first, the sketch of the workflow model is depicted by PN_1. Note that PN_1 is a WA WF-net and the tasks, ‘*handle_questionnaire*’ and ‘*processing*’, need further refinement. Then we use two WA WF-nets, PN_2 and PN_3, to replace these two tasks (transitions). In PN_3, the task ‘*process_until_OK*’ is a special transition;

we need an iteration of several tasks -- ‘*process_complaint*’, ‘*check_processing*’ and ‘*process_NOK*’ -- to represent it. Here we use PN_4, the extended net of a WA WF-net, to model this iteration. After these steps, there are no more special transitions that need further refinement and we get a sound workflow model depicted by a WRI WF-net.

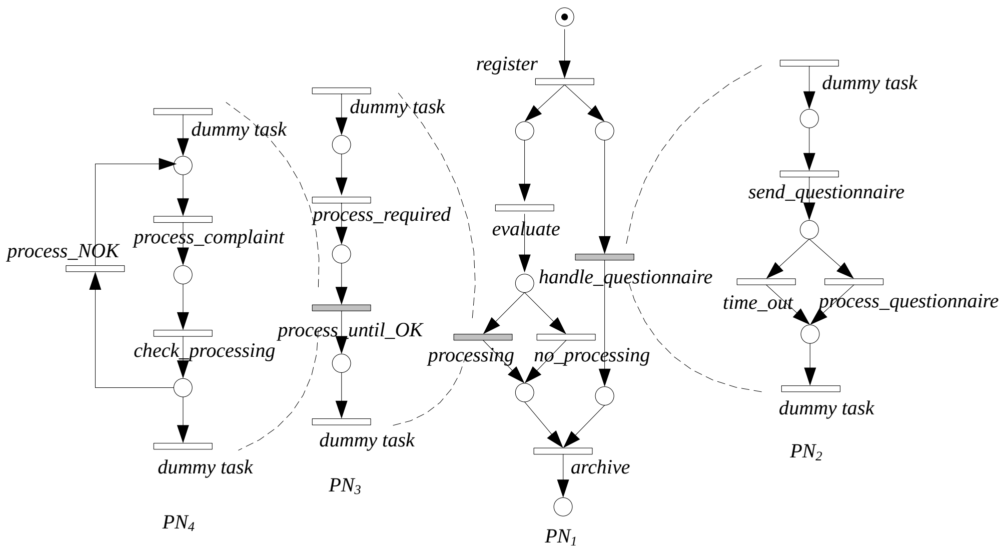


Figure 5.3 A hierarchical WRI-net for the processing of complaints

The major strength of the above modeling style lies in two aspects. (1) It ensures that the soundness property is preserved through WF-nets composition: For WF-nets, 1-soundness is not compositional. Even 1-soundness is maintained when we use a 1-sound WF-net to replace a transition of another 1-sound and safe one [1], the result WF-net in not necessarily safe which prevents us from making more compositions. Since the composition results in the above modeling procedure is WRI WF-nets, we can ensure the soundness of them. (2) At each step of modeling, the verification task is relatively simple: There exists a polynomial time algorithm to check whether a WF-net is a WRI WF-net, but we do not include it in this paper. We think that the desired way to use WRI WF-nets is to construct WF-nets by composition but not to create a WF-net at will and to check whether it is a WRI WF-net. During each step of constructing, the WF-net we use is often rather simple; we can even check whether they are WRI WF-nets manually.

WRI WF-nets do not fit for all workflow models. When complex synchronizations exist in workflow models, it may be hard to use WRI WF-nets to model them. The following example in Figure 5.4 adds a synchronization before the task *process_complaint* so that it can be invoked only if task *time_out* or *process_questionnaire* is finished. However, if we try to model it as in Figure 5.3, we must add arcs between PN_1 and PN_4 and the result would not be a WRI WF-net.

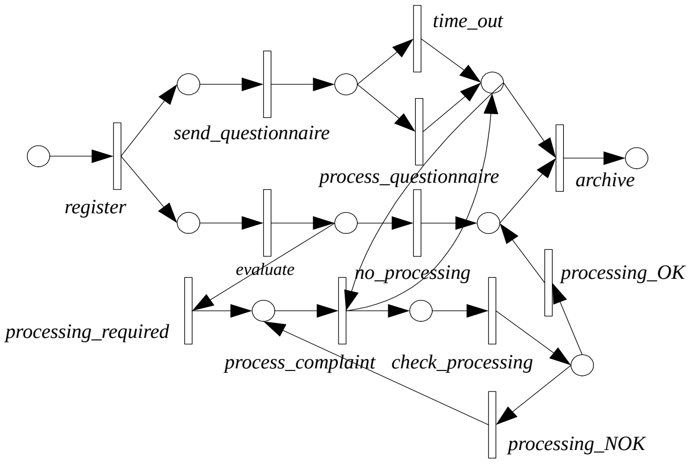


Figure 5.4 A workflow model that is hard to model by WRI WF-net

6 Conclusion

In this paper, we examined the relationship between 1-soundness, *k*-soundness and soundness of WF-nets. We first give some intuitive properties about the relation of 1-soundness and *k*-soundness, along with the *k*-soundness preservation during workflow composition. Then we proved that for several kinds of WF-nets, their 1-soundness implies soundness. Among these WF-nets, the FC WF-nets and WH WF-nets have particular practical value. We also introduced a special kind of WF-nets – WRI WF-nets with balanced “split” “merge” nodes and regular iterations which are inherently sound. In many cases, modeling problem could be solved by them.

Future work For more general kinds of WF-nets, it is still a question whether their 1-soundness could imply soundness. For example, it seems that for the AC WF-nets (WF-nets which are asymmetric choice) which is a superset of ST-AC WF-nets, the relationship between 1-soundness and soundness also holds, but we have not proved it yet. Other important properties of WRI WF-nets, e.g. efficient algorithm for determining reachability, are still under investigation.

Acknowledgements We want to thank Professor Natalia Sidorova of Department of Mathematics and Computer Science, Eindhoven University of Technology and Professor Dianxiang Xu of Department of Computer Science, North Dakota State University for their valuable suggestion.

Reference

[1] W. van der Aalst. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In W. van der Aalst, J. Desel, and A. Oberweis (Eds.), Business Process Managements: Models, Techniques, and Empirical Studies, Vol. 1806 of Lecture Notes in Computer Science, pages 161-183. Springer, 2000.

[2] K. van Hee, N. Sidorova, and M. Voorhoeve. Soundness and Separability of Workflow Nets in the Stepwise

Refinement Approach. In W. van der Aalst, E. Best (Eds), Application and Theory of Petri Nets 2003, Vol. 2678 of Lecture Notes in Computer Science, pp. 337-356. Springer, 2003.

[3] K. van Hee, N. Sidorova, and M. Voorhoeve. Generalised Soundness of Workflow Nets is Decidable. In J. Cortadella, W. Reisig (Eds), Application and Theory of Petri Nets 2004, Vol. 3099 of Lecture Notes in Computer Science, pp. 197-216. Springer, 2004.

[4] J. Desel, J. Esparza. Free choice Petri nets. Cambridge University Press, Cambridge, 1995.

[5] L. Jiao, T. Cheung, and W. Lu. On Liveness and Boundedness of Asymmetric Choice Nets. In Theoretical Computer Science, Vol. 311, pp. 165-197, 2004.

[6] K.Barkaoui, J.M.Couvreur, and C.Dutheillet. On Liveness in Extended non Self-Controlling Nets. In G. D. Michelis and M. D. Diaz (Eds), Application and Theory of Petri Nets 1995, Vol. 935 of Lecture Notes in Computer Science, pp. 25-44. Springer, 1995.

[7] J. Esparza, M. Silva. Circuits, Handles, Bridges and Nets. In G. Rozenberg (Ed), Advances in Petri Nets 1990, Vol. 483 of Lecture Notes in Computer Science, pp. 210-242. Springer, 1990.

[8] W. Sadiq, M. E. Orlowska. Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models. In M. Jarke and A. Oberweis (Eds), In Proc. of CAISE ’99, Vol. 1626 of Lecture Notes in Computer Science, pp. 195-209. Springer, 1999.

[9] W. Sadiq, M. E. Orlowska. On Correctness Issues in Conceptual Modeling of Workflows. In Proc. ECIS ’97.

[10] T. Murata. Petri Nets: Properties, Analysis and Applications. In Proc. IEEE Vol. 77(4), pp. 541-580, 1989.

[11] X. Lin, H. Hu etc. Design and Implementation of Routine Mechanism in CPMS. In Computer Science Vol. 31(3) 2004.

[12] K. Barkaoui, L. Petrucci. Structural Analysis of Workflow Nets with Shared Resources. In W. van der Aalst, G. D. Michelis and C. A. Ellis (Eds), WFM ’98, Vol. 98(7) of Computing Science Reports, Lisbon 1998.

[13] M. Hack. Analysis of production schemata by Petri nets. TR-94, MIT, Project MAC, Boston 1972.

[14] J. Desel. A Proof of the Rank Theorem for Extended Free Choice Nets. In K. Jensen (Ed.), Application and Theory of Petri Nets, Vol. 616 of Lecture Notes in Computer Science, pp 134-183. Springer, 1992.

[15] J. Esparza, M. Silva. On the Analysis and Synthesis of Free Choice Systems. In G. Rozenberg (Ed.), Advances in Petri Nets, Vol. 483 of Lecture Notes in Computer Science, pp. 243-286. Springer, 1990.

Component-Based Behavioural Modelling with High-Level Petri Nets

Rémi Bastide, Eric Barboni
LIIHS – IRIT – Université Toulouse 1
Place Anatole France, 31042 Toulouse CEDEX, France
{bastide, barboni }@irit.fr

Abstract:
This paper presents a component model inspired by the CORBA Component Model, and an associated formal notation based on Petri nets and dedicated to the modelling of concurrent and distributed components. The model is illustrated by a case study that illustrates its hierarchical features, and shows how the main features of components can be mapped to the constructs of the Petri net.

Keywords:
Component-based programming, Petri nets, behavioural specification.

1 Introduction

The main contenders in the domain of industrial software component models currently appear to be the Microsoft .Net framework [11], java-based components such as JavaBeans [4] and the Corba Component Model (CCM) [14]. Although they differ widely in the details, they have settled on a core of common concepts, which indicates that this domain has reached some maturity. These common concepts are:

- Considering a component as a black box that is accessed through exposed software interfaces. These interfaces define the contract offered by the component,
- Providing for multicast, event-based communication as well as for unicast method invocation,
- Providing for the design-time assembly and configuration of components, and in particular,
- Design-time configuration of components through exposed properties.

Component-based programming emerged mainly thanks to the software industry, which was concerned in improving the reusability of software artefacts. It is now firmly established as a commercial market. A lot of work has been devoted to practical usability concerns in an industrial setting, such as deployment facilities for example.

Much less work has been devoted ahead of time to the formal and theoretical foundations of component-based programming. The research community, witnessing the industrial success of component-based programming, has started in the last few years devoting a lot of activity to laying out such foundations, notably through workshops such as MOCA [9] or symposiums such as FMCO [5].

This paper presents a formal model of components, and in particular aims at providing a formal semantics framework for the main concepts of software components as stated above. To this aim, we define five steps:

- Define a component model. We have chosen a component model strongly inspired by the Corba Component Model (CCM), yet simpler and more precise. In particular, we focus on a behavioural semantics of component activity, and leave out many practical aspects of CCM (such as deployment) that are fundamental in an industrial setting, but mainly resort to "plumbing" and convey little theoretical interest.

- Propose a notation to formally specify the internal behaviour of a software component. Our formal approach is based on High-Level Petri nets [6], and builds upon our previous work on formal specification of Corba objects [1]. It may be considered as an extension of this previous, object-oriented work to component-oriented programming. Its Petri net foundations makes it particularly well suited to the modelling of concurrent, distributed or event-driven systems [2], and amenable to formal verification.
- Define a mapping from the constructs of the component models (e.g. facets, receptacles, event sources and sinks) to the constructs of our Petri-net based behavioural formalism (e.g. places, transitions, etc.).
- Provide a formal definition of inter-components communication primitives, (invocation of methods, event-based communication and access to properties). This definition is also given in terms of Petri nets.
- Provide a denotational semantics of an assembly of components, in order to define the behaviour of such a system in terms of the individual behaviour of each component and of the formal definition of inter-component communication primitives.

The expected benefits of such an approach are threefold:

- Offer a convenient notation for describing the internal behaviour of concurrent and distributed components,
- Provide a formal, unambiguous semantics of component features such as event multicast or properties, especially inter-components communication,
- And, with the previous two being necessary conditions, offer some means to reason about assemblies of components designed with this approach, in particular to mathematically verify properties on them.

For space reasons, all five aspects cannot be presented in depth within this paper. We will focus on presenting the component model, and, through a case study, illustrate how our notation can be used to model the internal behaviour of software components. We thus leave out the formal semantics of inter-components communication, and the verification of models.

2 CompoNets, our component model

Our component model is called CompoNets (a contraction of *Component* and *Petri nets*). Its features are strongly inspired by CORBA CCM [14]. In particular, we follow the CCM philosophy to treat the features required by a component on par with the features it offers to other components.

A CompoNet presents to the external world an *Enveloppe* made of several *Ports*, through which other components will be able to communicate (Figure 1). Ports may be of different categories (*Facet*, *Receptacle*, *Event Source*, *Event Sink* or *Property*), but each port is defined by a pair (Name, Java interface[1]).

[1] The choice of using Java interfaces instead of some more abstract and technology independent notation may be a little surprising. Actually, the current version of our object-oriented tool, PetShop [13], uses CORBA-IDL interfaces. The existence of such a programming-language neutral Interface Definition Language might be a benefit in an industrial setting, where access to legacy applications and interoperability is a major concern. However, we work in a research setting where our primary concern is to explore formal component-based specifications, and we have found that using CORBA-IDL was for us a hindrance rather than a benefit: it forced us to first think in terms of the CORBA-IDL, and then to work on the java mapping of these interfaces when it came to using our (java-based) tool. Java interfaces being expressive enough for our purpose, we have chosen to use them directly, thus avoiding the cumbersome mapping. Although we use the Java syntax to describe software interfaces, our model is not dependent on Java and could be easily adapted to other languages for interface definition. The implementation of our

- **Facets**: a facet represents a set of functional features offered to other components. These features are represented by the methods in the Java interface associated to the facet. The same Java interface may be used in different facets, with different names.
- **Receptacles**: a receptacle represents a set of functional features that are required by the CompoNet to fulfil its function. Like for a facet, these features are represented by the methods of the associated Java interface.
- **Event Sources**: an event source describes a set of semantically related events that may be emitted by the CompoNet. The fact that a CompoNet offers an event source implies that it will be able to multicast these events to a set of receivers who have manifested their interest for these events. Event sources are also described by a Java *Event interface*, each of the methods in this interface representing one particular event. Event interfaces have two syntactic constraints: 1) they must be derived from the standard java interface `java.util.EventListener`. 2) The methods they specify must return no result (i.e., in Java, return `void`), so that they can be called asynchronously, and must take one single argument of type `java.util.EventObject` (actually any subclass thereof, as allowed by the Java semantics of polymorphism). These constraints are essentially the same as the one used in the JavaBeans framework.
- **Event Sinks**: an event sink describes a set of events that the CompoNet is willing to receive. The definition of an event sink is the same as for an event source, with the same constraints.
- **Properties**: a CompoNet offers a set of configurable properties, which affect somehow its behaviour, and are meant to be specified at design time, when the component is combined with other to form an executable sub-system. Their value will usually be some form of constant (e.g. an instance of `java.awt.Color`). While all other ports are specified using Java interfaces, properties are defined with Java classes, or primitive types (`integer`, `boolean`, etc.), since an actual value for them has to be provided by assembly tools. They have no graphic representation.

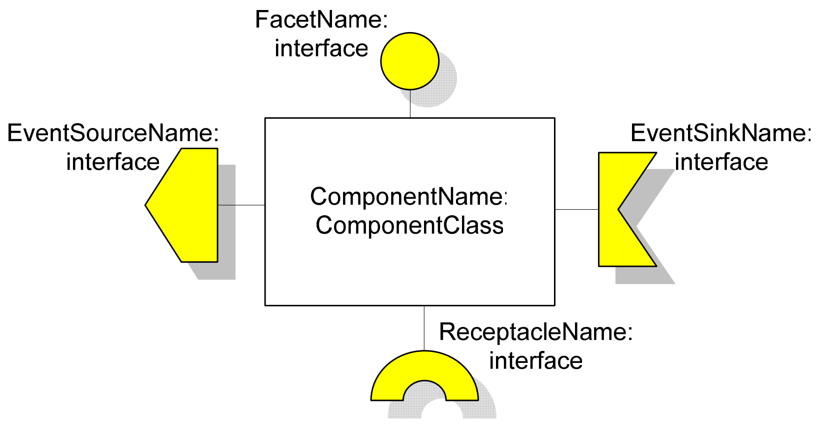


Figure 1: Graphic syntax of CompoNets

2.1 Ports of a CompoNet: a gesture recognizer

To illustrate various types of ports offered by a CompoNet, we will use a simple but realistic example. We want to describe as a CompoNet a piece of software encapsulating several 2D gesture recognition algorithms à la Rubine [12]. Such a component handles low-level mouse or

---

model that we are developing, on the other hand, is Java-based and aims at an easy interoperability with other Java technologies, mainly the Swing library for user interfaces.

pen-generated events, and generates higher level events when a specific gesture (e.g. a gesture for cut, copy or duplicate) is recognized.

```
public interface MouseListener
extends EventListener {
   public void mouseClicked(MouseEvent e);
   public void mousePressed(MouseEvent e);
   public void mouseReleased(MouseEvent e);
   public void mouseEntered(MouseEvent e);
   public void mouseExited(MouseEvent e);
}
```

```
public interface MouseMotionListener
extends EventListener {
   public void mouseDragged(MouseEvent e);
   public void mouseMoved(MouseEvent e);
}
```

```
public interface MouseInputListener
extends MouseListener,MouseMotionListener {
}
```

```
public interface ActionListener
extends EventListener {
   public void actionPerformed(ActionEvent e);
}
```

```
public interface GestureRecognizer {
   public void start();
   public void stop();
}
```

Figure 2: Java interfaces of the GestureRecognizer component

Figure 2 illustrates the Java interfaces used to describe the *Gesture Recognizer* CompoNet. One of them (`GestureRecognizer`) has been defined ad hoc, while the other ones come straight from the Java standard libraries. `MouseListener` and `MouseMotionListener` describe common mouse events and are combined for convenience into `MouseInputListener`, while `ActionListener` is an "all purpose" event interface commonly used to notify that "something interesting" has happened.

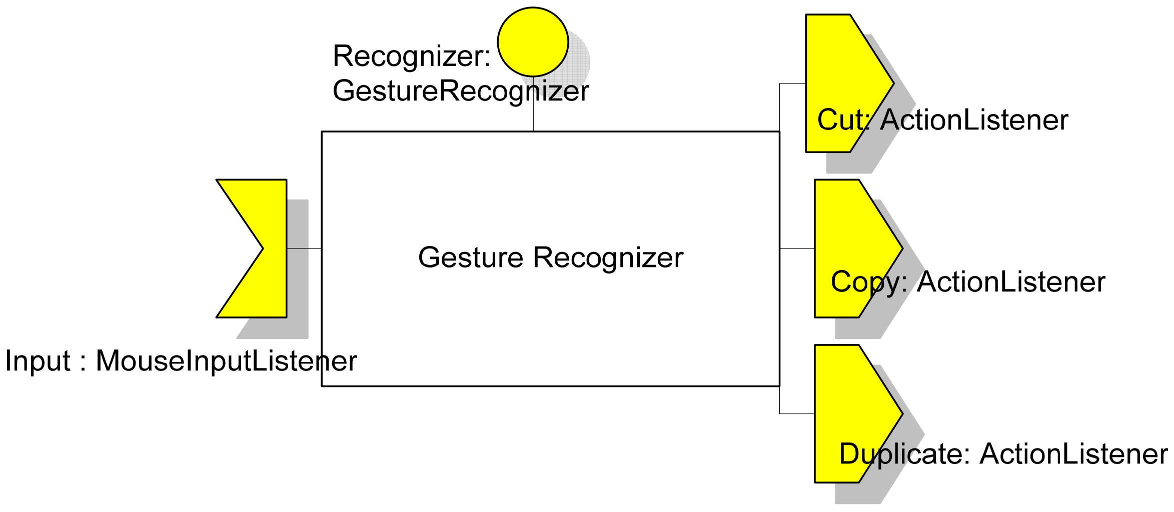


Figure 3: Ports of the GestureRecognizer component

The CompoNet in Figure 3 specifies that a *Gesture Recognizer* needs to receive mouse events (the event sink named "*Input*"), and eventually produces *actionPerformed* events (trough the *ActionListener* interface) when specific gestures (in this case *Cut*, *Copy* or *Duplicate*) are recognized. This is modelled by the three event sources of the same names. It also presents the facet *Recognizer* associated with the interface *GestureRecognizer*, which allows for starting and stopping the recognition.

2.2 Assemblies: wiring CompoNets together

CompoNets treat the ports offered by a component (facets and event sources) on par with the ports they require (receptacles and event sinks). This allows combining several components by wiring together compatible ports, thus forming a sub-system of linked components. A receptacle (resp. event source) is compatible with a facet (resp. event sink) if the Java interface associated to the latter is assignable to the one of the former, with the usual Java polymorphism semantics. Such a sub-system of wired CompoNets is called an *Assembly*.
For example, the Gesture Recognizer in Figure 3 could be wired into an executable Java application by linking it to a Java component that triggers mouse events, ant to another one that consumes *actionPerformed* events.

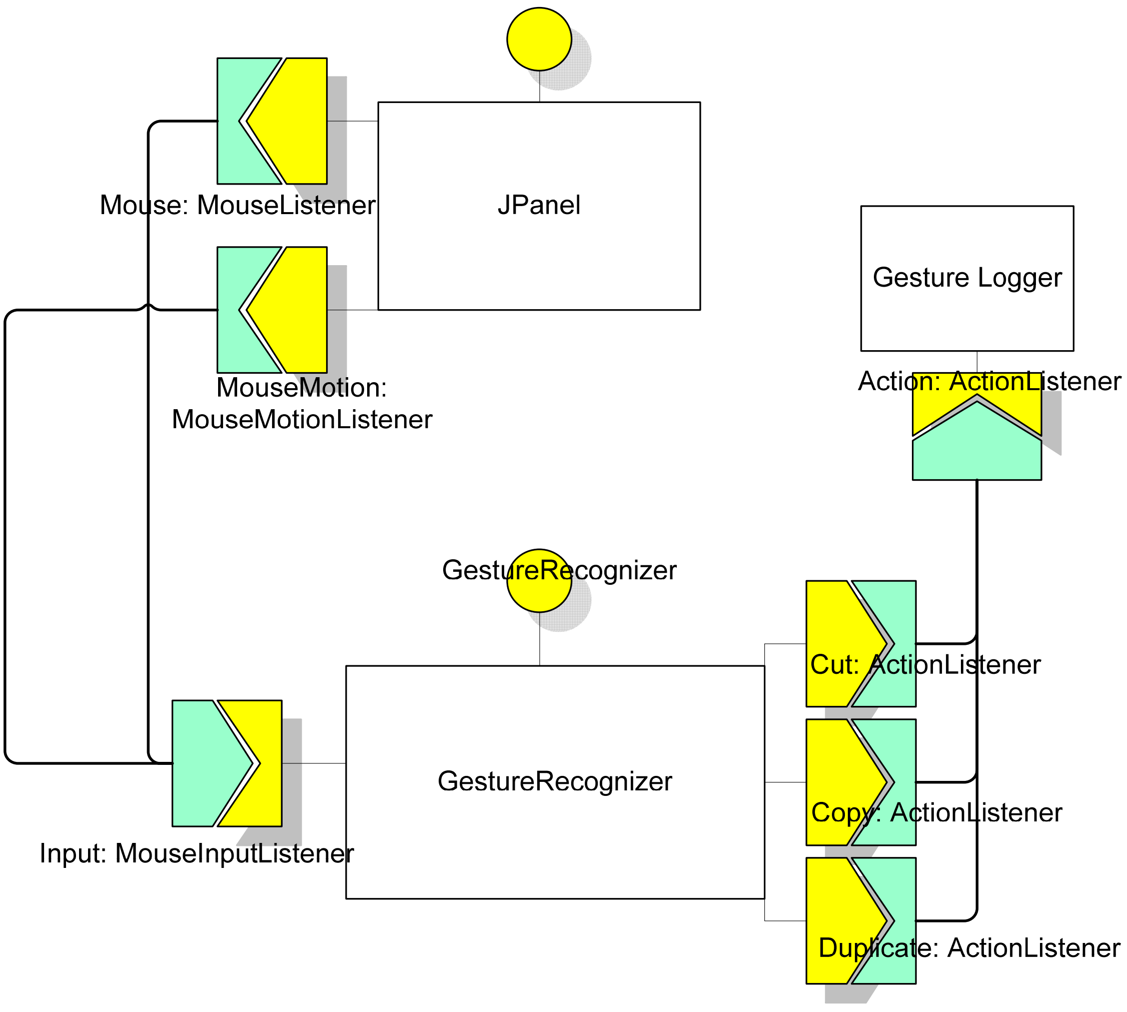


Figure 4: an Assembly using the Gesture Recognizer

In Figure 4 we use a standard Swing component (*JPanel*) that is a source of mouse events. The event sources it features could be automatically detected using the Java introspection mechanism. It's *Mouse* and *MouseMotion* event sources can be wired to the *Input* event sink of the *Gesture Recognizer* since their interfaces are compatible in the Java sense. We can also postulate the existence of a *Gesture Logger* component that accepts *actionPerformed* events and logs them somewhere as they occur. Such a component can be wired to the event sources of the *Gesture Recognizer* as illustrated by Figure 4. The hierarchic structure of CompoNets (i.e. the ability to define a CompoNet as a multi-level hierarchy of other CompoNets) will be further described in §3.

3 Case study: Head Gesture Recognition

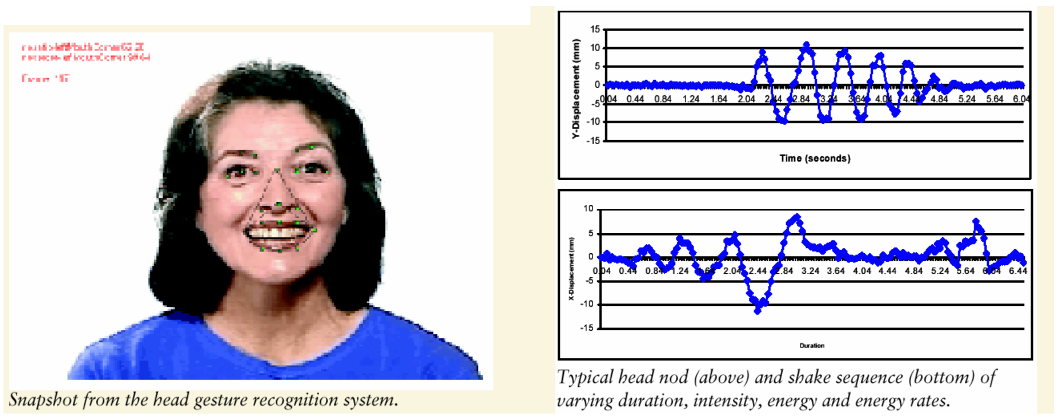


Figure 5: The head gesture recognition system

The case study used to illustrate our approach comes from the domain of Human-Computer Interaction. It uses computer vision and image recognition to allow the user to answer to yes/no dialogues using head nods and shakes as an alternative to clicking on yes/no buttons in a conventional dialog box. This study was presented in [7] without reference to component-based programming, and using finite state machines as a modelling tool.

We can provide a component-oriented view of this system using the CompoNet notation (Figure 6).
This model features 3 sub-components:

- *VideoAnalyser* : the role of this component is to process each video frame in order, and to detect moves of the user's head on the horizontal an vertical axes. It features an event sink (*NextVideoFrame*) that will allow it to be notified when a new video frame is available for processing. It also features two event sources (*DeltaX* and *DeltaY*) that it will use to notify that a move has been detected along the horizontal axis (*DeltaX*) or the vertical axis (*DeltaY*). These three ports are all defined using the standard java *ActionListener* event interface (Figure 2).
- Two instances of the *LinearMovementDetector* component. The role of this component is to detect a series of moves along the same axis. To this end, this component has two event sinks (*MainAxis* and *OtherAxis*) through which it will be informed that a movement has been detected along one or the other axis. When a linear movement has been detected, the component will notify this through its event source *LinearMove*. It is interesting to notice the connexions between *VideoAnalyser* and the two instances of *LinearMovementDetector*, which are identical except for reversing the roles of the two axes.

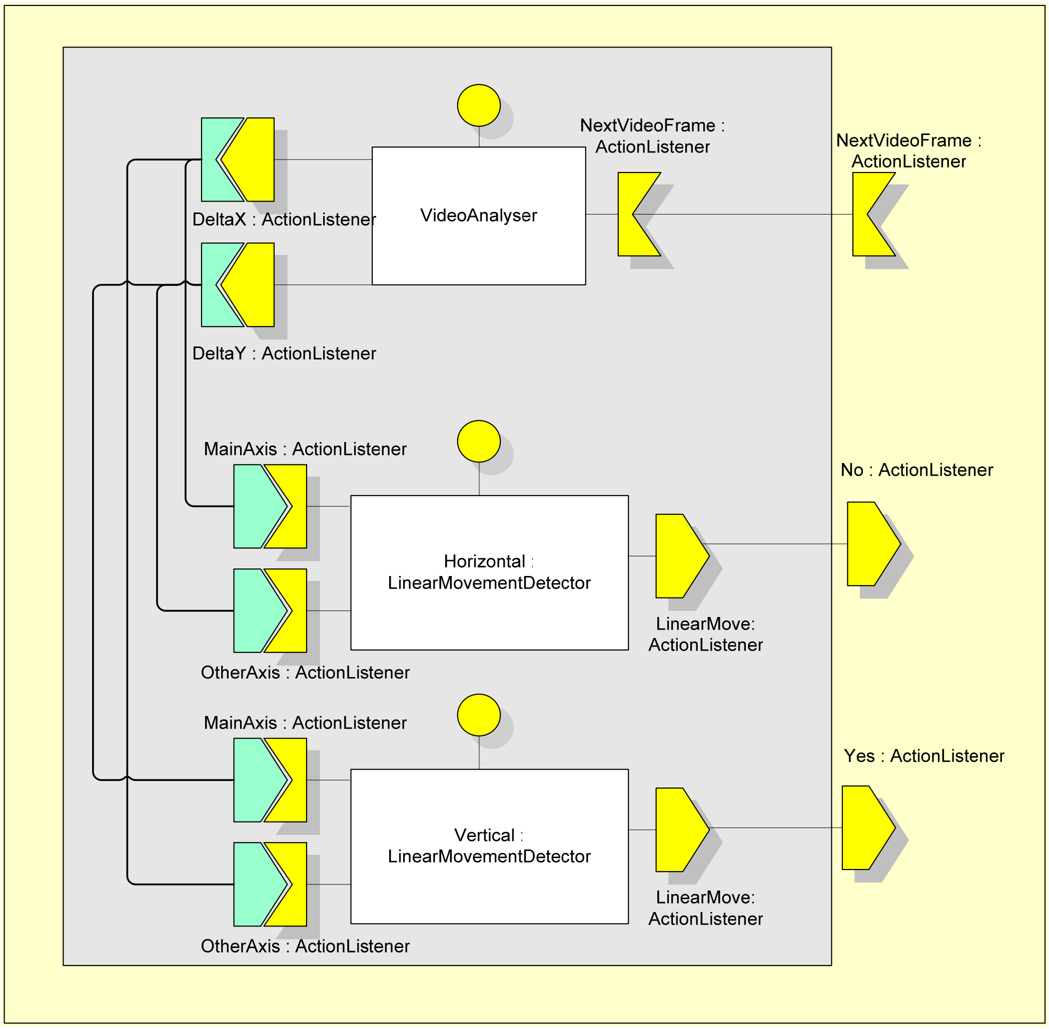

NextVideoFrame : ActionListener
NextVideoFrame : ActionListener
DeltaX : ActionListener
VideoAnalyser
DeltaY : ActionListener
MainAxis : ActionListener
No : ActionListener
Horizontal : LinearMovementDetector
LinearMove: ActionListener
OtherAxis : ActionListener
MainAxis : ActionListener
Yes : ActionListener
Vertical : LinearMovementDetector
LinearMove: ActionListener
OtherAxis : ActionListener

Figure 6: the CompoNet assembly

The hierarchical structure of CompoNets is illustrated on Figure 6 and Figure 7, the latter being the hierarchical view of the former. Hierarchy consists in hiding, promoting and renaming features at the upper level. Here, for examples, the two *LinearMove* event sources are promoted and renamed *Yes* and *No*. According to the axis considered, this series of moves will be considered as a head nod (*Yes*) or shake (*No*). NodAndShakeRecognizer can be reused as a black-box component in other assemblies, without caring for its internal structure.

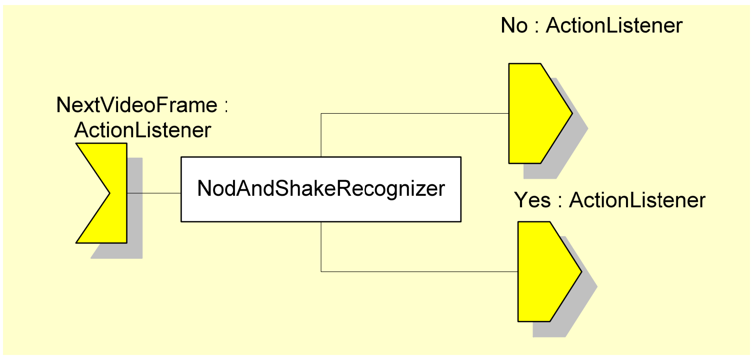


Figure 7: Hierarchical view of the NodAndShakeRegognizer

4 Formal Behaviour Specification

The notation we use for specifying the behaviour of components is based on high-level Petri nets. The main idea is to define a mapping between the features of the component model as described in §2 and the constructs of the Petri net.
We have developed this notation in previous work, to specify the behaviour of CORBA objects [3]. This previous work was concerned by augmenting CORBA-IDL interfaces with a behavioural specification based on Petri nets. The basic CORBA object model is based on unicast synchronous inter-object communication through the invocation of methods. The signatures of methods supported by a CORBA object are described in a programming-language independent notation, CORBA-IDL. In [3] and [1], we have shown how a Petri net (called the ObCS for Object Control Structure) can be used to specify the internal behaviour of an object conformant to a CORBA-IDL interface. The semantics of inter-object communication is also given in terms of Petri nets, using the well-known "client-server" Petri net pattern first described in [10]. Having both the internal behaviour and the communication protocol described in terms of Petri nets enables us to provide a formal semantics in a "denotational" way, i.e. the semantics of an object system of objects is given by an algorithm, that, given in input the ObCS of all classes in the system and the initial system configuration, can produce a single, unstructured high-level Petri net, which models the behaviour of the system as a whole. In our new CompoNet model, CORBA-IDL interfaces are superseded by Facet / Receptacles but the principles remain basically the same.

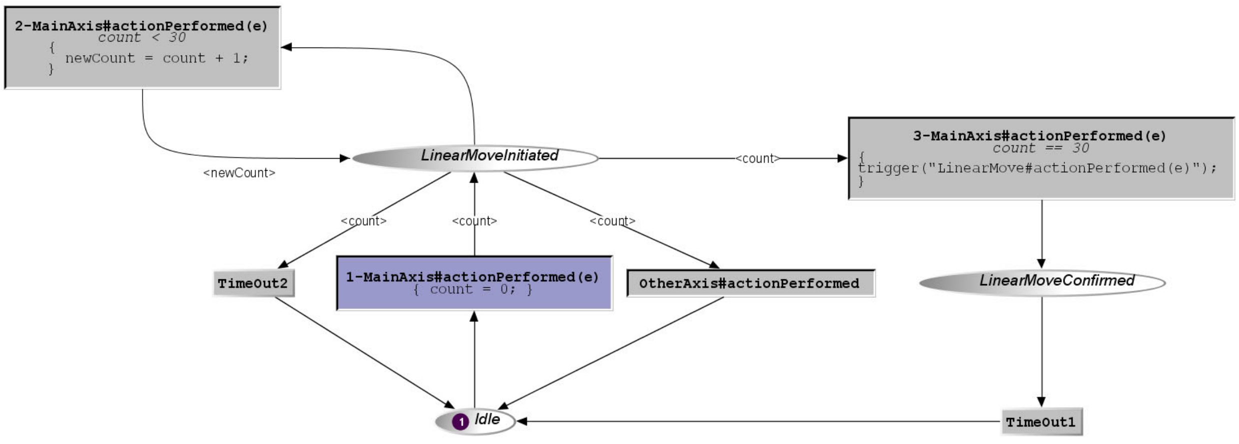
The main innovation of the CompoNet model is the introduction of multicast asynchronous event-based communication through event sources and sinks.

Mapping for event sinks:
Event sinks model set of events that will be received by the component, originating from other components in the system. In the Petri net that models the behaviour of a CompoNet, we can define **synchronized transitions** [8] that will occur when such an event is received, if they are enabled in the Petri net. Syntactically, transitions are associated to incoming events by their names, e.g. in Figure 8 all transitions whose name is in the form `X-`*`MainAxis#ActionPerformed`*`(e)` are related to the event *ActionPerformed* in the event sink called *MainAxis*. *e* is the event variable associated with the event. When an event occurs and no associated transition is enabled, the event is simply ignored.
Mapping for event sources
Event sources model the availability to multicast events to all interested receivers. In the Petri net, an event source is represented by the possibility to trigger events as a result of the occurrence of any transition. Syntactically, this is model by an action in the transition of the form: **`trigger`** *`EventSourceName#EventName`*`(`*`EventParameter`*`).`

4.1 Specification of the LinearMovementDetector

The formal specification of the component class LinearMovementDetector is given in Figure 8 as a Petri net using the mapping described above.
This net can be described informally as follows: It tries to detect a series of movements on its main axis (this is modelled by the 3 transitions *X-MainAxis#ActionPerformed*. In this example, we suppose that a series of 30 movements along the same axis will be considered as a nod or shake, because the head cannot possibly move infinitely along one axis, but is forced to reverse its movement. When this series has been detected, an event is triggered through the *LinearMove* event source (Transition *3-MainAxis#ActionPerformed*). If a movement is detected along the other axis, the detection is reset (transition *otherAxis#ActionPerformed*). Conversely, if a timeout occurs during detection (the head stands still for a given amount of time), the detection is also reset (Transition *TimeOut2*). Finally, after a linear move has been detected, some timeout

is also applied before starting the new recognition, to prevent several linear movements to be detected if the user shakes her head too long (Transition *TimeOut1*).



```
2-MainAxis#actionPerformed(e)
        count < 30
    {
      newCount = count + 1;
    }
```

```
3-MainAxis#actionPerformed(e)
        count == 30
{
trigger("LinearMove#actionPerformed(e)");
}
```

```
1-MainAxis#actionPerformed(e)
        { count = 0; }
```

Figure 8: Petri net of the LinearMovementDetector

5 Conclusion and future work

We have presented some aspects of our formal model of components, called CompoNets, mainly the component model itself and the Petri net-based notation used to specify the internal behaviour of components. We have presented a small but illustrative case study, showing in particular the hierarchical aspects of the model.

In previous work, we have developed a software environment for the design, analysis and interpretation of our CORBA-based object-oriented models [13]. We are now working on a component-based environment supporting our new model. In particular we aim at providing a lot of flexibility for the designer, allowing her to seamlessly use Petri nets or plain Java to implement the behaviour of components. For instance, in our case study, the LinearMovementDetector component is elegantly modelled as a Petri net because of its event-driven and time-driven nature, while the VideoAnalyzer component, which is mainly algorithmic, would more simply be expressed in Java. This leaves the opportunity for the designer to use the tool best suited to the problem, and to perform formal analysis on parts of the system that particularly deserve it.

6 Bibliography

1. Bastide, Rémi, Palanque, Philippe, Sy, Ousmane, Le, Duc-Hoa, and Navarre, David. "Petri-Net Based Behavioural Specification of CORBA Systems." *20th International Conference on Applications and Theory of Petri Nets, ICATPN'99* , Williamsburg, VA, USA. Susanna Donatelli, and Jetty Kleijn, Volume editors. Lecture Notes in Computer Science, no. 1639. Springer (1999) 66-85 .

2. Bastide, Rémi, Sy, Ousmane, Navarre, David, and Palanque, Philippe. "A Formal Specification of the CORBA Event Service." *IFIP TC6/WG6.1 4th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*, Stanford university, California, USA. Scott F. Smith, and Carolyn F. Talcott, editors . Kluwer (2000) 371-96.

3. Bastide, Rémi, Sy, Ousmane, and Palanque, Philippe. "Formal Specification and Prototyping of CORBA Systems." *13th European Conference on Object-Oriented Programming, ECOOP'99*, Lisbon, Portugal. Rachid Guerraoui, Volume editor. Lecture Notes in Computer Science, no. 1628. Springer (1999) 474-94.

4. Englander, Robert. *Developing Java Beans*. O'Reilly (1997).

5. FMCO. "Workshop on Modelling of Objects, Components, and Agents." Web page, Available at http://www.daimi.au.dk/CPnets/workshop01/.

6. Jensen, Kurt. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. 2nd edition ed., Vol. 2 Springer-Verlag (1996).

7. Kapoor, Ashish, and Picard, Rosalind W. "A Real-Time Head Nod and Shake Detector." *Workshop on Perceptive User Interfaces*, Orlando, Florida. ACM International Conference Proceeding Series, ACM Press (2001)

8. Moalla, M, Pulou, J, and Sifakis, J. "Synchronized Petri Nets: a Model for the Description of Non-Autonomous Systems.". Lecture Notes in Computer Science, ed. Springer-Verlag, no. 64. (1978) 374-84.

9. MOCA. " Second International Symposium on Formal Methods for Components and Objects." Web page, Available at http://fmco.liacs.nl/fmco03.html.

10. Ramamoorthy, C. V., and Ho, G. S. "Performance Evaluation of Asynchronous Concurrent Systems." *IEEE Transactions of Software Enginnering* 6, no. 5 (1980) 440-449.

11. Richter, Jeffreys. *Programming the .NET Platform*. Microsoft Press (1998).

12. Rubine, Dean. "Combining Gestures and Direct Manipulation." *ACM SIGCHI Conference on Human Factors for Computing Systems (CHI)*, Monterey, California. ACM Press 659-60.

13. Sy, Ousmane, Bastide, Rémi, Palanque, Philippe, Le, Duc-Hoa, and Navarre, David. "PetShop: a CASE Tool for the Petri Net Based Specification and Prototyping of CORBA Systems." *20th International Conference on Applications and Theory of Petri Nets, ICATPN'99*, Williamsburg, VA, USA. (1999)

14. Vinoski, Steve. "New Features for CORBA 3.0." *Communications of the ACM* 41, no. 10 (1998) 44-52.

Dynamic Analysis Algorithm for ECATNets

Noura Boudiaf * and Allaoua Chaoui**

***Laboratoire Lire, Université de Constantine, Algérie**

or

Département d'Informatique, Université d'Oum El Bouaghi, Algérie

Fax : 00 213 32 42 23 85

boudiafn@yahoo.com

****Department of Software Engineering, Faculty of IT,**

University of Philadelphia, Jordan

chaoui@philadelphia.edu.jo

Abstract. ECATNets are a category of algebraic Petri nets based on a sound combination of algebraic abstract types and high level Petri nets [Bet92]. The most distinctive feature of ECATNets is that their semantics is defined in terms of rewriting logic [Mes95], allowing us to built models by formal reasoning. Automated techniques such as model checking [Bcm92] are efficient for specifying and verifying finite-state models. However, they are not suitable for specifying infinite-state systems. The rewriting logic language Maude allows model checking only for finite-state system. The verification based model checking in ECATNet is possible thanks to their integration in Maude. In the objective to extend this possibility to deal with infinite-state system, we propose in this paper dynamic analysis for ECATNet based on construction of their reachability graph for infinite-state model as well as finite-state one. Dynamic analysis is well defined for standard Petri nets and high-level Petri nets. The lack of any work related to dynamic analysis in order to profit effectively of ECATNets advantages motivated us to perform this work. Such proposition allows in the future the development of dynamic analysis tool by using Maude system.

Keywords : ECATNet, Rewriting Logic, Maude, Dynamic Analysis, finite-state systems, infinite-state systems.

1. Introduction

ECATNets [Bet92] are a kind of net/data model combining the strengths of Petri nets with those of abstract data types. The most distinctive feature of ECATNets is that their semantics is defined in terms of rewriting logics [Mes96], allowing us to built models by formal reasoning. Motivating ECATNets (*Extended Concurrent Algebraic Term Nets*) leads to motivating Petri nets, abstract data types, as well as their association into a unified framework. Petri nets are used for their foundation in concurrency and dynamics, while abstract data types are used for their data abstraction power and solid theoretical foundation. Their association into a unified framework is motivated by the need to explicitly specify process behavior and complex data structures in real systems. ECATNets have been used in the area of manufacturing systems [Bet93, Bet94, Bet97]. In addition of the modeling, the ECATNets permit already validation and the simulation of concurrent programs [Bet96, Bet97]. In this paper, we look at the side of verification.

Automated techniques such as model checking [Bcm92] are efficient for specifying and verifying finite-state models. However, they are not suitable for specifying most infinite-state systems. The rewriting logic language Maude [Mes92, Man99] implements LTL (*linear Temporal logic*) model checking [Mcc03]. This one allows only the verification of finite-state systems. The fact that ECATNet formalism is integrated [Bel00] in rewriting logic and so in its language Maude, benefit of the possibility of the verification based on model checking of a system described by using ECATNet. However, as told above, it is not possible to do the verification of liveness proprieties for example for infinite-state models. In the goal of extending these possibilities of verification for ECATNet formalism, we propose in this paper reachability analysis for ECATNet based on construction of their reachability graph for infinite-state model as well as finite-state one. For it, we study first the case of unbounded places. The creation of dynamic analysis tool of the ECATNets is foreseeable by using Maude system. Such tool is a basic element to construct an exhaustive environment for ECATNet.

1.1 Paper Organization

The remainder of this paper is organized as follows: the section 2 is a general presentation of the ECATNets and their description in rewriting logic. A detailed study of non-bounded places case is presented in the section 3. The section 4 is dedicated to the proposition of recovery of the infinite increase of numbers of elements of a multi-set. Section 5 contains our proposed algorithm of ECATNets dynamic analysis. In section 6, an example of an ECATNet and the construction of its reachability graph are given. Section 7 concludes the paper.

2. ECATNets

ECATNets [Bet92] are a kind of net/data model combining the strengths of Petri nets with those of abstract data types. The most distinctive feature of ECATNets is that their semantic is defined in terms of rewriting logics [Mes96], allowing us to built models by formal reasoning. Motivating ECATNets (Extended Concurrent Algebraic Term Nets) [Bet91] leads to motivating Petri nets, abstract data types, as well as their association into a unified framework. Petri nets are used for their foundation in concurrency and dynamics, while abstract data types are used for their data abstraction power and solid theoretical foundation. Their association into a unified framework is motivated by the need to explicitly specify process behavior and complex data structures in real systems. ECATNets have been used in the area of manufacturing systems [Bet93, Bet94, Mao97].

2.1 The Syntax

From a syntactic point of view, places are marked with multisets of algebraic terms. Input arcs of each transition t ,i.e. (p,t), are labeled by two inscriptions IC(p,t) (Input Conditions) and DT(p,t) Destroyed Tokens), output arcs of each transition t, i.e. (t,p'), are labeled by CT(t,p') (Created Tokens), and finally each transition t is labeled by TC(t) (Transition Conditions). IC(p,t) specifies the enabling condition of the transition t, DT(p,t) specifies the tokens (a multiset) which have to be removed from p when t is fired, CT(t,p') specifies the tokens (a multiset) which have to be added to p' when t is fired. Finally, TC(t) represents a boolean term which specifies an additional enabling condition for the transition t. The current ECATNets state is given by the union of terms having the following form (p, M(p)). As an example, the distributed state s of a net having one transition t and one input place p marked by the multiset $a \oplus b \oplus c$, and an empty output place p', is given by the following multiset : $s = (p, a \oplus b \oplus c) \otimes (p', \phi)$.

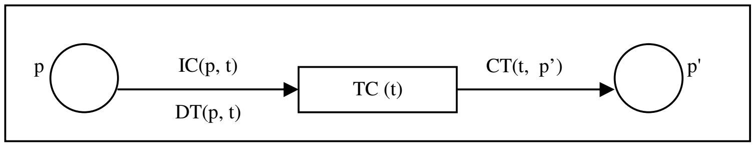


Fig 1. *A generic ECATNet*

2.2 The Semantic

It is worthwhile to mention that it is not easy to explain the behavior of ECATNets merely by giving the equivalent of a firing-like rule [Bet92]. This is because of their level of abstraction as well as their concurrent behavior. However, this behavior may be informally commented in the following way. A transition t is enabled when various conditions are simultaneously true. The first condition is that every IC(p,t) for each input place p is enabled. The second condition is that TC(t) is true. Finally the addition of CT(t,p') to each output place p' must not result in p' exceeding its capacity when this capacity is finite. When t is fired DT(p,t) is removed from the input place p and simultaneously CT(t,p') is added to the output place p'. Transition firing and its conditions are formally expressed by rewrite rules which are strongly depending on the form of the syntactic notation used for representing IC(p,t). Such rules act in reality as the axioms of the rewrite logic mentioned above. The basic notions of this logic in relation of its use with ECATNets are summarized as follows. The axioms are in reality conditional rewriting rules describing transitions effects as elementary types of changes. The deduction rules allow us to draw valid conclusions about the evolution of the ECATNet from these changes. A rewrite rule is a structure of the form "t: u $\rightarrow$ v if boolexp"; where u and v are respectively the left and the right-hand sides of the rule, t is the transition associated with this rule, and boolexp is a Boolean term. More precisely u and v are multisets of pairs of the form (p, $[m]_{\oplus}$), where p is a place of the net, $[m]_{\oplus}$ a multiset of algebraic terms, and the multiset union on these terms, when the terms are considered as singletons. The multiset union on the pairs ((p, $[m]_{\oplus}$) will be denoted $\otimes$. We let $[x]_{\otimes}$ denote the equivalence class of x, w.r.t. the ACI (*Associativity Commutativity Identity*) axioms for $\otimes$. An ECATNet state is itself represented by a multiset of such pairs where each place p is found at least once. Given a set R of rewriting rules (defining all the elementary types of changes), we say that R entails a sequent s $\rightarrow$ s' (defining a global change from a state s to a state s') iff s $\rightarrow$ s' can be obtained by finite and concurrent applications of the following rules of deduction: Reflexivity, Congruence, Replacement [Mes 96]; Splitting, Recombination, and Identity [Bet92]. The Reflexivity rule says that everything may be transformed into itself. The Congruence rule says that elementary changes have to be correctly propagated. The Replacement rule is used when variable, instantiations are necessary. The Splitting and Recombination rules allow us, by "judiciously" splitting and recombining different multisets of equivalence classes of terms, to detect ECATNet computations exhibiting a maximum of parallelism. The Identity rule allows to relate ϕ_B (i.e., the identity element of $\oplus$ with ϕ_B(i.e., the identity element of $\otimes$. We now recall the forms of the rewrite rules (i.e., the metarules) to associate with the transitions of a given ECATNet.

IC(p,t) is of the form $[m]_{\oplus}$

case1 $[IC(p,t)]_{\oplus} = [DT(p,t)]_{\oplus}$

The form of the rule is then given by:

t : (p, $[IC(p,t]_{\oplus} \rightarrow$ (p', $[CT(t,p')]_{\oplus}$

where t is the involved transition, p its input place, and p' its output place.

case2 $[IC(p,t)]_{\oplus} \cap [DT(p,t)]_{\oplus} = \phi_M$

This situation corresponds to checking that IC(p,t) is included in M(p) and, in the positive case, removing DT(p,t) from M(p). In the case where DT(p,t) is not included in M(p), we have to remove the elements which are common to these two multisets.

The form of the rule is given by:

t : (p, $[IC(p,t)]_{\oplus} \otimes$ (p, $[DT(p,t)]_{\oplus} \cap [M(p)]_{\oplus} \rightarrow$ (p, $[IC(p,t)]_{\oplus} \otimes$ (p',$[CT(t,p')]_{\oplus}$)

Case3 $[IC(p,t)] \cap [DT(p,t)] \neq \phi_M$

This situation corresponds to the most general case. It may however be solved in an elegant way by remarking that it could be brought to the two already treated cases. This is achieved by replacing the transition falling into this case by two transitions which, when fired concurrently, give the same global effect as our transition (see [Bet92] for more detail). In reality, this replacement shows how ECATNets allow to specify a given situation at two levels of abstraction. The forms of the axioms associated with the extensions are, w.r.t. the explanation already given, evident and thus not commented.

IC(p,t) is of the form $\tilde{}[m]_{\oplus}$

The form of the rule is given by:

t : (p, $[DT(p,t)]_{\oplus} \cap M(p)\,]_{\oplus} \rightarrow$ (p',$[CT(t,p')]_{\oplus}$) if ($[IC(p,t]_{\oplus} \setminus ([IC(p,t)]_{\oplus} \cap [M(p)]_{\oplus}) = \phi_M \rightarrow$ [false]

IC(p,t) = empty

The form of the rule is given by:

t: (p, $[DT(p,t]_{\oplus} \cap [M(p) \cap) \rightarrow$ (p',$[CT(t,p') \cap$) if ($[M(p) \cap \rightarrow \phi_M$

When the place capacity C(p) is finite, the conditional part of the rewrite rule will include the following component:

($[CT(p,t)]_{\oplus} \oplus [M(p]_{\oplus}) \cap [C(p)]_{\oplus} \rightarrow [CT(p,t)]_{\oplus} \oplus [M(p)]\,]_{\oplus}$ (**Cap**)

In the case where there is a transition condition TC(t), the conditional part of our rewrite rule must contain the following component: TC(t) $\rightarrow$ [true] (see [Bet 92] for more details).

3. Study of Unbounded Places Case

For a dynamic analysis of an ECATNet, the construction of reachability graph is foreseeable. The development of an algorithm that detects and covers all cases of unbounded places in an ECATNet is a delicate problem. It is about an unbounded place when the number of algebraic terms in this place increases infinitely.

The study of case of an unbounded place comes back to study monotony property. In an ECATNet, this property depends strongly on assignments of algebraic term variable that label arcs joining places and transitions. We separate three cases of the ECATNets : Simple ECATNet (without conditions of transitions and ECATNet's places with infinite capacity), ECATNet with conditions of transitions and places with infinite capacity, ECATNet without conditions of transitions and places with infinite capacity, and of course the general case.

In the case of ECATNet with places with infinite capacity, the property of monotony is respected. In the case of the places with finite capacity, the monotony may exist or not.

We focus in our study on the case when IC(p,t) is of the form $[m]_{\oplus}$ and we exclude from the two other cases (IC(p,t) is of the form $\tilde{}[m]_{\oplus}$, and IC(p,t) = empty).

3.1 ECATNet's Places with Infinite Capacity

We have in following some propositions and their proofs. We focus in these propositions on the ECATNet of the first case ($[IC(p,t)]_{\oplus} = [DT(p,t)]_{\oplus}$). We obtained the same result for the two other cases ($IC(p,t)]_{\oplus} \cap [DT(p,t)]_{\oplus} = \phi_M$ and $[IC(p,t)] \cap [DT(p,t)] \neq \phi_M$).

3.1.1 Absence of Transitions Conditions

Proposition 1. Let M , M' two markings and S a sequence of transitions ; if $M \xrightarrow{S}$ and $M \subseteq M'$ then $M' \xrightarrow{S}$

Proof 1. We make call to the proof by recurrence. For $S = t$ (one transition), if t is enabled since M , then we have :

$\forall p \in P \quad IC(p,t) \subseteq M(p) \quad \text{or} \quad DT(p,t) \subseteq M(p)$

$\forall p \in P \quad \text{if} \quad M(p) \subseteq M'(p) \quad \text{then} \quad IC(p,t) \subseteq M'(p) \quad \text{or} \quad DT(p,t) \subseteq M'(p)$

Consequently, t is enabled at M' . Let's assume that this property is verified for $S = t_1..t_k$ and we prove that it is for $S = t_1..t_k t_{k+1}$. We have:

$M \xrightarrow{t_1..t_k} M_k \xrightarrow{t_{k+1}} M_{k+1}$

By supposition, $M \subseteq M'$ then $M' \xrightarrow{t_1..t_k} M'_k$

Know, is t_{k+1} enabled since M'_k ?

We have $M \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \ldots \xrightarrow{t_k} M_k$

$\forall p \in P \qquad M_1(p) = (M(p) \setminus DT(p,t_1)) \cup CT(p,t_1)$

If $IC(p,t_1) \subseteq M(p)$ et $DT(p,t_1) \subseteq M(p)$

Such that \ and $\cup$ are subtraction and union of multi-sets. While $DT(p,t_1) \subseteq M(p)$ then, without risks in multi-sets, we can write :

$M_1(p) = (M(p) \cup CT(p,t_1)) \setminus DT(p,t_1)$

…

$M_k(p) = (M_{k-1}(p) \cup CT(p,t_{k-1})) \setminus DT(p,t_{k-1})$

Then :

$M_k(p) = ((M_{k-2}(p) \cup CT(p,t_{k-2})) \setminus DT(p,t_{k-2})) \cup CT(p,t_{k-1}) \setminus DT(p,t_{k-1})$

We can write :

$$M_k(p) = ((M_{k-2}(p) \cup CT(p,t_{k-2}) \cup CT(p,t_{k-1})) \setminus (DT(p,t_{k-2}) \cup DT(p,t_{k-1}))$$

Consequently, we can write :

$$M_k(p) = ((M(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l))) \setminus (\bigcup_{l=1}^{k} DT(p,t_l))$$

Moreover, we have another side :

$$M'_k(p) = ((M'(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l))) \setminus (\bigcup_{l=1}^{k} DT(p,t_l))$$

If $M_k(p) \subseteq M'_k(p)$ then $M(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l)) \subseteq M'(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l))$

And then

$$(M(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l))) \setminus (\bigcup_{l=1}^{k} DT(p,t_l)) \subseteq (M'(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l))) \setminus (\bigcup_{l=1}^{k} DT(p,t_l))$$

Because $\bigcup_{l=1}^{k} DT(p,t_l) \subseteq M(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l))$ that is to say $M_k(p) \subseteq M_k^{'}(p)$

If t_{k+1} is enabled since M_k then it is enabled since $M_k^{'}$.

If $M \xrightarrow{t_1..t_{k+1}}$ and $M \subseteq M'$ then $M^{'} \xrightarrow{t_1..t_{k+1}}$. It means that the property of monotony is verified.

Proposition 2. If $M \xrightarrow{s} M_1$ and $M \subseteq M_1$ then p place, such that $M(p) \subset M_1(p)$ is an unbounded place.

Proof 2. We have in this case :

$M \xrightarrow{s} M_1 ... \xrightarrow{s} M_k$ When k offers toward the infinite with $M \subseteq M_1$ $M_1 \subseteq M_2$ and $M_{k-1} \subseteq M_k$

For $p \in P$ if $M(p) \subset M_1(p)$ then $\exists m \neq \phi$ $M_1(p) = m \cup M(p)$ (m is non-empty multi-set). On the other hand $M_2(p) = ((M_1(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l))) \setminus (\bigcup_{l=1}^{k} DT(p,t_l))$

Consequently, $M_2(p) = ((m \cup M(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l))) \setminus (\bigcup_{l=1}^{k} DT(p,t_l))$

Without risk, we write $M(p) = m \cup ((M(p) \cup (\bigcup_{l=1}^{k} CT(p,t_l))) \setminus (\bigcup_{l=1}^{k} DT(p,t_l))$

Which means that $M_2(p) = m \cup M_1(p)$ or $M_2(p) = m \cup m \cup M(p)$

We have $m \subset m \cup m$ and then $M(p) \subset M_1(p) \subset M_2(p)$

By recurrence, we will have $M(p) \subset M_1(p) \subset M_2(p) \subset ... \subset M_k(p)$ or $M_k(p) = \underbrace{m \cup ... \cup m}_{k \ \ time} \cup M(p)$ That is to say, that if k offers toward the infinite, then the number of the algebraic terms in place p increase toward the infinite.

Interpretation 2. For one transition *t*, we have :

$\forall p \in {}^{\bullet}t \quad M^{'}(p) = M(p) \setminus DT(p,t)$ if $IC(p,t) \subseteq M(p)$

$\forall p \in t^{\bullet} \quad M^{'}(p) = M(p) \cup CT(p,t)$

For $p \in t^{\bullet}$ $M(p) \subseteq M^{'}(p)$ then $M(p) \subseteq M^{'}(p) \cup CT(p,t)$

It is achieved some either circumstances.

$p \in {}^{\bullet}t$ $M(p) \subseteq M^{'}(p)$ then $M(p) \subseteq M^{'}(p) \setminus DT(p,t)$

It is possible, only in two cases :

- $DT(p,t) = \not\subset$, we sensitize without withdrawing
- input place is always output place $DT(p,t) \subseteq CT(p,t)$

We add more algebraic terms than we have just withdrawn.

3.1.2 Presence of Transitions Conditions

The presence of a condition for a transition does not make any problems with regard to the preservation of monotony. A condition of a transition is true if values that give back it true are inside its input places. While increasing multi-sets in these places, these values always exist and the condition is always true. That wants to say, if a transition is enabled since a marking M , it is always asince M' such that $M \subseteq M'$.

3.2 ECATNet's Places with Finite Capacity

We distinguish tow cases. We discuss them in the following propositions.

Proposition 3. if $M \xrightarrow{S} M'$ and $M \subseteq M'$ and for every finite place *p*, *M(p) = M'(p)*, then every infinite place *p'* such $M(p') \subset M'(p')$ is an unbounded place.

Proof 3. Immediate.

Proposition 4. if $M \xrightarrow{S} M'$ and $M \subseteq M'$ and the first transition in S become not enabled since M'. If it exists *S'* such that $M' \xrightarrow{S'} M''$ S' stops when S become enabled. If we have the following case :

$M \xrightarrow{S} M' \xrightarrow{S'} M'' \xrightarrow{S} M'''$ and if $M' \subseteq M'''$ and $M''(p) \subseteq M(p)$ for each place *p* with bounded capacity yielding S disabled at M', then every infinite place *p'* such $M'(p') \subset M'''(p')$ is an unbounded place.

Interpretation 4. The alone reason that makes the first transition of *S* disabled is the problem of the overtaking of certain places capacities. *S'* makes lower the algebraic terms in places suffering of overflow. For it, S becomes again enabled, then we get M'''. Then, we get the infinity in certain places with infinite capacities.

4. Recovery of the Infinite Increase of Elements Numbers in Multi-set

For every multi-set whose elements are of type T, we create ω_T which covers the increase of number of elements in this multi-set. Being given m a multi-set whose elements are of type T , then : $\forall m : T \quad m \cup \omega_T = \omega_T$, $m \subseteq \omega_T$, $\omega_T \setminus m = \omega_T$, $\omega_T \subseteq \omega_T$, $\omega_T \cup \omega_T = \omega_T$

Such $\setminus$, $\subseteq$, $\cup$ are in the same way the classic operation extension symbol on the multi-sets. The new operations apply on the union of the composed whole of all multi-sets of elements of type T and the new multi-set ω_T.

We don't need the operation $\oplus$, the operation $\otimes$ is sufficient while basing on the concept of decomposition. If a place contains many algebraic terms, for example $(p, a \oplus b)$, than we can write it as $(p, a) \otimes (p, b)$ when a, b are elementary algebraic terms.

We interpret previous ideas concerning the recovery of the infinite increase of elements numbers in a multi-set by using rewriting logic :

- $(p, \omega_T) \otimes (p, \omega_T) = (p, \omega_T)$
- $(p, e) \otimes (p, \omega_T) = (p, \omega_T)$ where e is an algebraic term of type T. By recurrence:
  $(p, e_1) \otimes \ldots \otimes (p, e_n) \otimes (p, \omega_T) = (p, \omega_T)$ where $e_1, \ldots, e_n$ are of type T and $n \geq 1$
- If we have a rewriting rule of the shape : $(p, e_1) \otimes \ldots \otimes (p, e_n) \otimes m \rightarrow m'$ if C (m doesn't concern p), then we add rewriting rule :
  $(p, \omega_T) \otimes m \rightarrow (p, \omega_T) \otimes m'$ if C.

5. An Algorithm to Construct Reachability Graph for ECATNets

First, we define some functions that will be used in the framework of our algorithm:

SubMarking(m, m') : returns true if m is included in m'.

StSubMarking(m, m') : returns true if m is sub-marking of m' and m $\neq$ m'.

ReachableMarking(m, l) : it is the function that returns a marking after have apply the rewriting rule l on the marking m. So ReachableMarking(m, l) = ϕ, then m doesn't have a successor while using the rule l.

CoveredPlaceInMarking(p, m) : for p place. It returns an marking that is sub-marking of m after have covering the sub-marking of m that concerns p. that is to say: if m = $(p, e_1) \otimes \ldots \otimes (p, e_n) \otimes m'$ (m' is independent from p and $n \geq 1$), then CoveredPlaceInMarking(p, m) = $(p, \omega_T) \otimes m'$

CoveredSetPlaceInMarking(P, m) : for a set of places P.

ProjectMarkingOnPlace(m, p) : it gives a marking that is sub-marking of m concerning the place p.

PlacesTobeCovered(m, m', IP, l) : this function returns a set of unbounded places E, a subset of IP. For every p of E, the function look if there is an effective augmentation of tokens in this place when comparing between m' and m. i.e, ProjectMarkingOnPlace(m', p) is strictly sub-marking of ProjectMarkingOnPlace(m, p). A small algorithm for this function is described as follows :

```
function PlacesTobeCovered(m, m', IP, l) : set of places;
var E, F : set of places;
E := {}; F := IP;
for p ∈ F do
if StSubMarking(ProjectMarkingOnPlace(m', p), ProjectMarkingOnPlace(m, p)) = true then
   begin E := E ∪ {p}; F:=F / {p}; end;

return(E);
```

Now, we give our proposed algorithm for the construction of reachability graph for ECATNet.

Algorithm *Construction of reachability graph for ECATNet*

Input *: ECATNet N without arcs inhibitor, P: set of places of N, P = FP$\cup$IP, FP : set of finite places and IP is the set of infinite places, FP$\cap$IP = ϕ. L : set of transitions (rewriting rules) of N.*

Output *: Finite reachability graph for N.*

```
Method :
var E : set of places
1. The root is labeled by the initial marking $m_0$
2. A marking m doesn't  have a successor if and only if:
        - for each rewriting rule l, ReachableMarking(m, l) = $\phi$
        - it exists on the path of $m_0$ to m another marking  m' = m
3. if the two conditions are not verified, let m'' be the marking  such $m \xrightarrow{l} m''$
  for each rule l, where ReachableMarking(m, l) $\neq \phi$ do
    if $\exists$ m' : marking on the path of $m_0$ until  m & m' is a sub-marking of ReachableMarking(m, l)
    then  if $\forall p \in FP$ :
            ProjectMarkingOnPlace(ReachableMarking(m, l), p)= ProjectMarkingOnPlace(m',p)
        then  E := PlacesTobeCovered(ReachableMarking(m, l), m', IP, l);
              m'' := CoveredSetPlaceInMarking(E, ReachableMarking(m, l));

        else    if   $\exists l_i$ (i =1, 2)$\in$ L &  $l_1 \neq l_2$ on the path from m' until  m  such that
                  &  $\exists$m1, m2  tow markings on the path  from  m' until m
                  &  $m' \xrightarrow{l_1 Sl} m_1$ and $m_1 \xrightarrow{l_2 S'} m_2$ and $m_2 \xrightarrow{l_1 Sl}$ ReachableMarking(m, l)
                  &  SubMarking(m',  m1) = true
                  &  ReachableMarking(m1, $l_1$) = $\phi$
                  &  $\forall p \in$ FP: ProjectMarkingOnPlace(m2, p)
                                  is sub-marking of ProjectMarkingOnPlace(m', p)

                then E  := PlacesTobeCovered(m, m', IP,l);
                     m'' = CoveredSetPlaceInMarking(E, ReachableMarking(m, l));
                endif;
        endif;
    else  m'' = ReachableMarking(m, l);
    endif;
```

6. Example

The subject of this section is the application of previous algorithm on simple industrial case. This example is presented in [Lin95] and it is described by using ECATNet formalism in [Mao97]. We take this description with some modifications. This example presents an infinite-state real system.

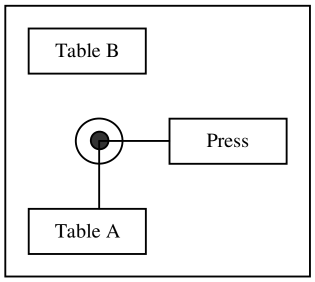


Fig 2. Production Cell

6.1 Example Presentation

The example is about a cell of production that manufactures forged pieces of metal with the help of a press. This cell is composed of a table *A* that serves to feed the cell by raw pieces, of a robot of handling, a press and a table *B* that serves to the storage of forged pieces. The robot includes two arms, disposed at right angles on one same horizontal plan, interdependent of one same axis of rotation and without vertical mobility possibility. The figure 2 represents the spatial disposition of elements of the cell. The robot can seize a raw piece of the table *A* and to put down it in the press with the help of the arm 1. It can also seize a forged piece of the press and can put down it on the table of storage *B* with the help of the arm 2. In short, the robot can do two movements of rotation. The first allows it to pass from its initial position to its secondary one. This movement permits the robot to deposit a raw piece in the press and possibly the one of a forged piece on the table of storage *B*. The second the fact to pass its secondary position toward its position initial and permits it to pursue the cycle of rotation.

6.2 ECATNet Model of the Example

Figure 3 represents the ECATNet model of production cell. The symbol ϕ is used to denote the empty multi-set in arcs inscriptions. Please note that *r* denotes 'raw' and *f* denotes 'forge'. If the inscriptions IC(p, t) and DT(p, t) are equals, then we present just IC(p,t) on the arc (p, t).

ECATNet Places.

Ta : table A ; set, possibly empty, of raw pieces.

Tb : table B ; set, possibly empty, of raw pieces.

Ar1 : arm 1 of robot ; at most a raw piece.

Ar2 : arm 2 of robot ; at most a forge piece.

Pr : press ; at most a raw piece or a forge piece.

Pos-I : initial spatial position of robot ; it is marked "ok" if it is the current position of robot.

Pos-S : secondary spatial position of robot ; it is marked "ok" if it is the current position of robot.

EA : this place is added for testing if the tow arms of robot are empty.

ECATNet Transitions.

T1 : Taking of a raw piece by the arm 1 of the robot.

T2 : Taking of a forge piece by the arm 2 of the robot.

D1 : deposit of a raw piece in the press.

D2 : deposit of a forge piece on the table B.

TS1, TS2 : rotation of the robot from its initial position towards its secondary position.

TI : rotation of the robot from its secondary position towards its initial position.

F : forge of the raw piece introduced in the press.

E : deposit of a raw piece on the table A.

R : removing forge pieces from the table A.

Rewriting Rules.

[T1] : (Ta, raw) $\rightarrow$ (Ar1, raw)   if (M(Pos-I) $\rightarrow$ ok)

and ((Ar1, raw) $\otimes$ M(Ar1) $\cap$ C(Ar1)) $\rightarrow$ (Ar1, raw) $\otimes$ M(Ar1)

[T2] : (Pr, forge) $\rightarrow$ (Ar2, forge)   if (M(Pos-I) $\rightarrow$ ok)

and ((Ar2, raw) $\otimes$ M(Ar2) $\cap$ C(Ar2)) $\rightarrow$ (Ar2, raw) $\otimes$ M(Ar2)

[D1] : (Ar1, raw) $\rightarrow$ (Pr, raw) $\otimes$ (Ea, Ear1)   if (M(Pos-S) $\rightarrow$ ok)

[D2] : (Ar2, forge) $\rightarrow$ (Tb, forge) $\otimes$ (Ea, Ear2)   if (M(Pos-S) $\rightarrow$ ok))

[TS1] : (Pos-I, ok) $\rightarrow$ (Pos-S, ok) if (M(Ar1) $\rightarrow$ raw)

[TS2] : (Pos-I, ok) $\rightarrow$ (Pos-S, ok) if (M(Ar2) $\rightarrow$ forge)

[TI] : (Pos-S, ok) $\otimes$ (Ea, Ear1) $\otimes$ (Ea, Ear2) $\rightarrow$ (Pos-I, ok)   if (M(Pos-S) $\rightarrow$ ok)

[F] : (Pr, raw) $\rightarrow$ (Pr, forge)

[E] : $\phi \rightarrow$ (Ta, raw)

[R] : (Tb, forge) $\rightarrow \phi$

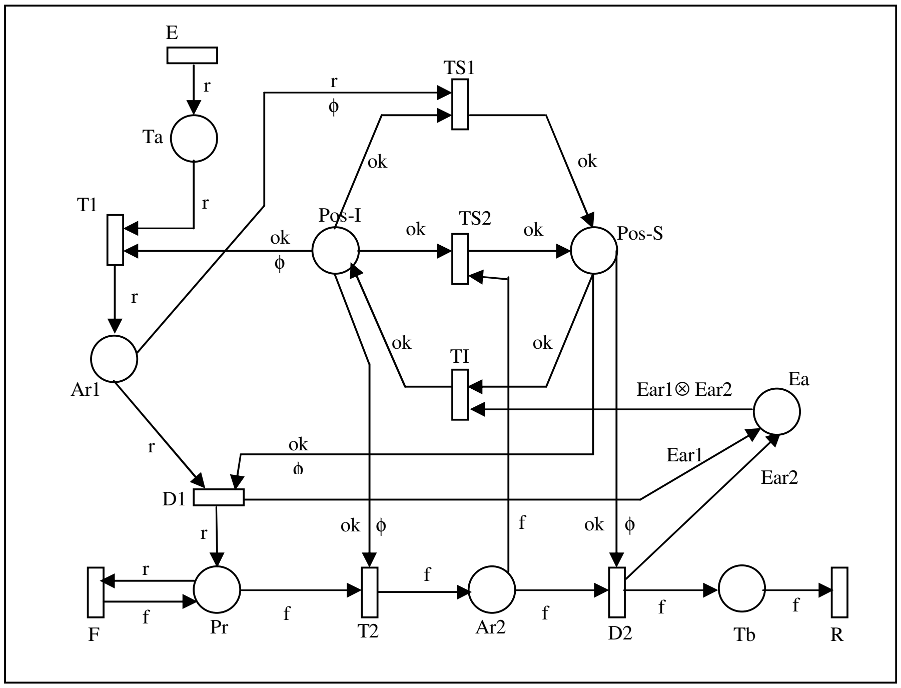


Fig 3. ECATNet Model of the cell

6.3 Reachability Analysis of the Example

For simplicity, we give just a part of reachability graph (one path) obtained by using previous algorithm. Initial marking is in line 1. The unique enabled transition is E. The firing of this transition since the initial marking makes the marking (Ta, r) ⊗ (Pos-I, ok) ⊗ (Ea, Ear2). We note that initial marking is included in this marking and places with finite capacity have constant marking along this path. The place Ta is unbounded, then we cover the infinite augmentation of markings in this place by using the symbol ω_{Raw}, when Raw is the enumerated type containing the value raw. At each accessible marking, the transition E is always enabled. The firing of E since such marking gives the same marking. The marking in line 7 is included in the marking of in the line 8. However, the place Pr is a finite place. The firing of the transition D2 is enabled since marking in line 11. The firing of this transition gives the marking : (Ta, ω_{Raw}) ⊗ (Pos-S, ok) ⊗ (Pr, r) ⊗ (Ea, Ear1) ⊗ (Ea, Ear2) ⊗ (Tb, forge). We note that the marking in line 5 is included in the one of line 12. In addition to that, for every finite place p, ProjectMarkingOnPlace(ReachableMarking(t_{11}, l), p)) is equal to ProjectMarkingOnPlace(t_5, p). By applying the previous algorithm, the place Tb is unbounded and we replace this obtained marking by one described in line 12.

1. $(Pos\text{-}I, ok) \otimes (Ea, Ear2) \xrightarrow{E}$

2. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}I, ok) \otimes (Ea, Ear2) \xrightarrow{T1}$

3. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}I, ok) \otimes (Ar1, r) \otimes (Ea, Ear2) \xrightarrow{TS1}$

4. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}S, ok) \otimes (Ar1, r) \otimes (Ea, Ear2) \xrightarrow{D1}$

5. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}S, ok) \otimes (Pr, r) \otimes (Ea, Ear1) \otimes (Ea, Ear2) \xrightarrow{F}$

6. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}S, ok) \otimes (Ea, Ear1) \otimes (Ea, Ear2) \otimes (Pr, f) \xrightarrow{TI}$

7. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}I, ok) \otimes (Pr, f) \xrightarrow{T1}$

8. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}I, ok) \otimes (Pr, f) \otimes (Ar1, r) \xrightarrow{T2}$

9. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}I, ok) \otimes (Ar2, f) \otimes (Ar1, r) \xrightarrow{TS1, TS2}$

10. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}S, ok) \otimes (Ar2, f) \otimes (Ar1, r) \xrightarrow{D1}$

11. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}S, ok) \otimes (Ar2, f) \otimes (Pr, r) \otimes (Ea, Ear1) \xrightarrow{D2}$

12. $(Ta, \omega_{Raw}) \otimes (Pos\text{-}S, ok) \otimes (Pr, r) \otimes (Ea, Ear1) \otimes (Ea, Ear2) \otimes (Tb, \omega_{Forge})$

7. Conclusion

In this paper, we presented a proposition of an algorithm to construct reachability graph for ECATNets. We exclude in this algorithm arcs inhibitor study. This proposition is motivated by the fact that model checking cannot deal with infinite-state model in terms of verification. In addition to that, unlike Petri nets and high level Petri nets, ECATNet formalism doesn't possess any dynamic analysis algorithm. This proposition permits the development of a dynamic analysis tool subsequently for the ECATNets. The development of a tool for the construction of reachability graph is not complicated. The construction of reacahbility graph for finite-state system is already developed by using Maude system. This is possible thanks to the reflectivity of Maude language. We can describe an ECATNet in Maude and this description become as input to another program written in Maude itself. Another interesting step in our work is to extend this algorithm to deal with ECATNets containing arcs inhibitor.

Bibliography

[Bcm92] J. R. Burch, E. M. Clarke, K. L. Mcmillan, D. L. Dill and L. H. Hwang. "Symbolic model checking". 10^{20} states and beyond. Information & Computation, 92(2) : 142-170, 1992.

[Bel00] F. Belala, M. Bettaz, L. Petruci. "Concurrent Systems Analysis Using ECATNets". Logic Journal of the IGPL, pp. 149-164, 2000.

[Bet92] M. Bettaz, M. Maouche. “How to specify Non Determinism and True Concurrency with Algebraic Term Nets” . Lecture Notes in Computer Science, N 655, Spring Verlag, Berlin, p. 11-30, 1992.

[Bet93] M. Bettaz, M. Maouche, M. Soualmi, and M. Boukebeche, “Protocol Specification Using ECATNets”, Networking and Distributed Computing, 7-35, 1993.

[Bet94] M. Bettaz, M. Maouche, M. Soualmi, M. Boukebeche. “On Reusing ATNets Modules in Protocol Specification”. Journal of Systems and Software 27(2), pp. 119-127, 1994.

[Bet96] M. Bettaz, A. Chaoui, K. Barkkaoui. “On Finding Structural Deadlocks in ECATNets Using a Logic of Concurrency”. Journal on Computing and Information, Vol 2 No 1, pp. 495-506, 1996.

[Bet97] M. Maouche, M. Bettaz, G. Berthelot, L. Petrucci. “Du Vrai Parallélisme dans les Réseaux Algébriques et de son Application dans les systèmes de Production. MOSIM, 1997.

[Lin95] T. Lindner. "Formal Development of Reactive Systems : Case Study Production Cell". In LNCS, N 891, Spring Verlag, Berlin, p. 7-15, 1995.

[Man99] M. Clavel and AL. “Maude : Specification and Programming in Rewriting Logic”. Internal report, SRI International, 1999.

[Mcc03] T. McCombs. “Maude 2.0 Primer, Version 1.0”. Internal report, SRI International, 2003.

[Mes92] J. Meseguer, “A Logical Theory of Concurrent Objects and its Realization in the Maude Language”. In G. Agha, P. Wegner, and A. Yonezawa, Editors, Research Directions in Object-Based Concurrency. MIT Press, 1992.

[Mes95] J. Meseguer, N. Marti-Oliet. “From Abstract Data Types to Logical Frameworks”. Invited, Paper, in Lecture Notes in Computer Science, N 906, pp. 48-80, 1995.

[Mao97] M. Maouche, M. Bettaz, G. Berthelot and L. Petrucci. “Du vrai parallélisme dans les réseaux algébriques et de son application dans les systèmes de production”. MOSIM’97, 1997.

Proposal of an Agent-based System for Distributed Software Development

Kolja Lehmann and Vanessa Markwardt

University of Hamburg, Faculty of Informatics
Vogt-Kölln-Str. 30, D-22527 Hamburg
`{8lehmann|9markwar}@informatik.uni-hamburg.de`

Abstract Heterogenous teams working geographically distributed on software projects present a challenge for the development of supporting development environments. Many tasks that are easy in a local context are much more difficult to handle in these environments, especially those of coordination and cooperation.
This paper outlines a possible architecture for an agent-based software system to assist with the development process. For the description of the agent interaction in this paper as well as for the actual implementation high-level Petri nets will be used.

Keywords: Agent, Software development, IDE, Mulan, nets within nets, Petri nets, Renew

1 Introduction

More and more software projects are concepted and implemented by teams of experts working geographically distributed. This creates new challenges for the organisation of the development processes as well as for the coordination of work and the collaboration towards a common goal. Traditional development environments are centered on the individual developer, sometimes integrating tools for version management. An integrated environment for distributed software development however must center on the aspect of collaboration among the whole team involved and facilitate the communication between team members.

This paper outlines a multi-agent system (MAS) for distributed software development. It is conceived as a collection of different tools, each one implemented as an agent, working together with each other and with the user of the system. The paper concentrates on the design of the system, using high-level Petri nets as a formalism for the design. The actual implementation will be covered in later publications. It is envisioned to use high-level Petri nets for the implementation as well, based on the works of Rölke [9] [8], Duvigneau [2] and others.

Section 2 will describe possible uses of the system. Section 3 will cover the different agents the system consists of and their interaction. Section 4 will outline the different agents and their interactions more detailed in a Petri net notation to show the patterns of communication to be used. Finally, section 5 summarises the results and shows the direction in which further work will be made to implement the system designed here.

2 Application Context

This section describes use cases for the MAS designed in the later sections. Therefore, some example applications will be given that can be integrated into the system along with one more detailed example used in the following sections to describe the architecture.

2.1 Usage Scenarios

We envision this system to be used as a complete development tool for distributed teams. All useful applications one could need will be available from within one interface. This involves single user tools like source code beautifier as well as groupware tools such as whiteboards, source code control, workflow management or discussion boards.

When working on a team, the project manager would assign a couple of tools to every team member, for example source code control and workflow execution, then special tools for different roles in the team like documentation or testing. Also, everyone might choose some tools they like or even build their own team specific tools to create their own individual IDE (Integrated Development Environment).

For example there could be agents used to schedule meetings and coordinate the calendars of the team members, autonomous agents performing automatic tests on the code base, which is maintained by another agent, that would also notify users about changes in files they are interested in.

2.2 Example: Whiteboard

An example application that will be used to explain the agent interaction envisioned is a whiteboard. This can be used for general discussion or collaborative design. It consists of one server that provides the whiteboard (essentially a text or graphic object). Every team member can then access the whiteboard through the system, read it, write comments or change the contents of the board.

Of course a whiteboard system is nothing new, nor is its client-server like implementation. It is only used here to show how different aspects of the system can work together. Instead of adding all kinds of functionality into one centralized server application, it will be possible to just add the server agent and a factory for the client agents to the system. Using mobile agents this can be highly scalable.

2.3 Comparison with other Solutions

In Computer supported collaborative work (CSCW) a lot of effort has been made to facilitate the working together of team members towards a common goal. Most of these systems use a client-server concept of organising the participants and the ressources of the system. Usually all data is stored on a central server to which all participants connect via some kind of client program [1].

This is only useful for collaboration within one enterprise or if one enterprise dominates the others in a cooperation. For virtual teams, assembled dynamically for each task, more flexible solutions are needed.

[5] describes a system for cross-organizational workflow, based on service contracts. This is not a multi-agent system, but based on traditional software components adhering to certain agreed-upon standards.

The system proposed here will allow a flexible combination of ressources to teams without a centralized controlling server. New participants can enter the system and offer new services or use existing ones in a flexible way.

3 Agent Interaction

This section covers the different agents that make up the system and their interaction. Every user of the system will be represented by a “user agent”, acting on his behalf and presenting the possible functions to the user.

The user agent does not provide functionality by itself but instead relies on different “tool agents” to provide different functions. The tool agents can be wrappers around legacy applications, provide useful functionality on their own or facilitate the communication with a server such as a file server or a whiteboard server or between tool agents. One tool agent will only be employed by one user agent at a time.

Figure 1 shows the composition of user and tool agents. The user agent registers itself with a tool agent, using a defined interface. The tool agent then provides functionality to the user agent. This connection is only virtual, because interaction between the agents only happens via message-passing. However, in some cases it might be beneficial if the tool agent resides locally close to the user agent so that shorter response times can be achieved. If the tool agent needs access to a user’s local files, it is necessary that it is located on the same machine as the user agent.

This shows how an agent can also act like an agent platform, hosting other agents and providing ressources to them. This housing of other agents can be virtual, a tool agent might even be plugged into several different user agents or it might be a physical moving of the tool agent into the execution environment of the user agent.

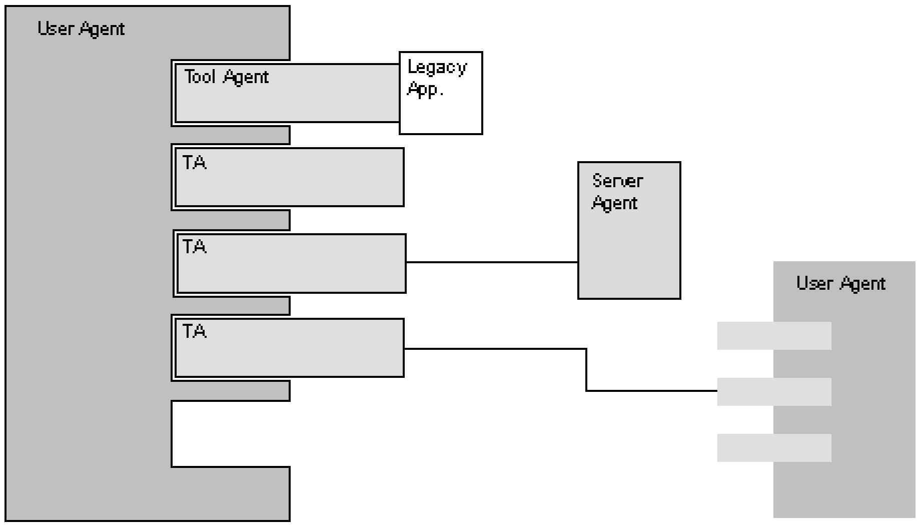


Figure1. Interaction between the different agents in the MAS

3.1 User Agents

The user agent is the users way of communication with the rest of the multi-agent system. It will provide some kind of graphical user interface and a possibility to add tool agents to its repertoire.

To add a new tool agent to a user agent, the tool agent must first be found, for example through some kind of directory service[1] or created for the user agent by another agent in the system. For example a whiteboard server agent could create a new whiteboard tool agent and send it to the requesting user agent instead of talking to the user agent directly.

3.2 Tool Agents

Tool agents provide functionality to users by plugging into a user agent. When binding to a user agent a tool agent will provide its user interface to the user agent, so that it can be displayed to the user. This can either be accomplished through a list of possible commands or through a special interface, e.g. a GUI-object to be integrated into the users' workspaces.

The agent then waits for requests from the user agent or may act proactively depending on the task it is meant for. This might involve acting as a wrapper around a legacy application, communicating with other tool agents or with other agents not directly involved with the system's users.

In the example of the whiteboard system, there would be one whiteboard server agent and several whiteboard tool agents, one for each user accessing the whiteboard. The server agent would not communicate directly with the user agents, but have the tool agents act instead. They would provide the functionality of the server, like "read", "write" to the user agent, in the simplest case just forwarding the requests to the server. To save bandwidth, the tool agent might also choose to only submit changes to the server instead of the whole new content.

3.3 Other Agents

Apart from these two types of agents there can also be others present in the system. This could be server agents, as mentioned in the whiteboard example, that offer services used by multiple tool agents, or services that work without user supervision, like agents for automated testing or documentation.

4 Petri Net Model

The interaction between the agents can be modelled with reference nets [6] as shown in figures 2 and 3. The net models here are only schematic and are not executable as they are right now. However, once the modelling gets more concrete the models will be executable in Renew [7].

The first net shows the user agent. Its interactions with the user are modelled by the transitions of the upper region of the net and the place "vision".

A user initiates a request to find a new agent for a specific task, which results in sending a request to the directory facilitator (DF). The DF sends a reply with a reference to an agent capable of providing the desired functionality. That agent is sent a "register" request and stored in the user agent's "TA's" place for future reference.

[1] In a FIPA-compliant MAS [4], this would be the directory facilitator agent

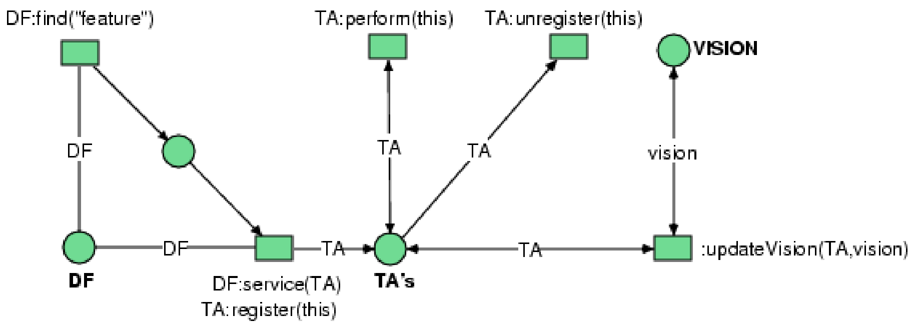


Figure2. Petri net model of the user agent

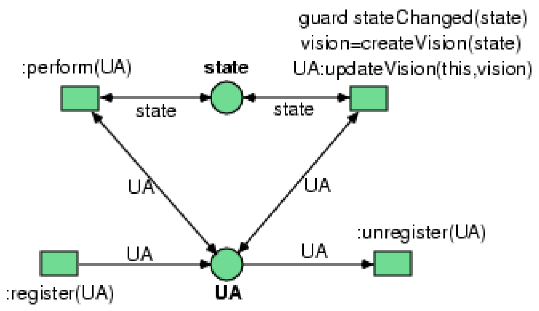

guard stateChanged(state)
vision=createVision(state)
UA:updateVision(this,vision)
:perform(UA)
state
state
state
UA
UA
:unregister(UA)
UA
UA
:register(UA)
UA

Figure3. Petri net model of the tool agent

The tool agents that are known to the user agent are registered in the place "TA's". The user agents can make calls to the tool agents, by use of the "perform" transition, gets state updates that reflect on the visional feedback the user gets and can unregister the agent.

The net on bottom shows the corresponding tool agent. It holds a reference to the user agent in the place "UA" and an internal state, however structured, in the place "state".

5 Discussion

This paper could only sketch out a basic plan for further work on implementing a MAS for distributed SW-development. The actual implementation along with a lot of details has been left open. Nevertheless, it has pointed out the direction of further research to be made in the next time, leading to a potentially very powerful distributed collaboration tool.

For the implementation of this MAS we will use the multi-agent platform Mulan[9] [8], which allows the modelling of agent interaction with reference nets while providing the complete infrastructure of agent creation, message passing, etc. With its extension Capa[3] we will also get FIPA-compliance and a directory service to build on.

Building on this foundation, we will build simple versions of the user agent along with one or two tool agents to experiment with different strategies of initialisation and functionality publishing. Later on other functions will have to be added, among others persistance and workflow management.

References

1. Conan Albrecht. A comparison of distributed groupware implementation environments. *Proceedings of the 36th Hawaii International Conference on System Sciences*, 2003.
2. Michael Duvigneau. Bereitstellung einer Agentenplattform für Petrinetzbasierte Agenten. Diplomarbeit, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, December 2002.
3. Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Concurrent architecture for a multi-agent platform. In *Agent-Oriented Software Engineering III: Third International Workshop, AOSE 2002, Bologna, Italy, July 15, 2002. Revised Papers and Invited Contributions*, number 2585 in Lecture Notes in Computer Science, pages 59–72, Berlin Heidelberg New York, 2003.
4. Foundation for intelligent physical agents. URL `http://www.fipa.org/`, 2004.
5. Paul W. P. J. Grefen, Karl Aberer, Heiko Ludwig, and Yigal Hoffner. Crossflow: Cross-organizational workflow management for service outsourcing in dynamic virtual enterprises. *IEEE Data Engineering Bulletin*, 24(1):52–57, 2000.
6. Olaf Kummer. *Referenznetze.* Dissertation, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2002.
7. Renew – the reference net workshop. URL `http://www.renew.de/`, 2004. Verweist auf Programme, Quelltexte und Dokumentation zum Petrinetzsimulator Renew.
8. Heiko Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen.* PhD thesis, University of Hamburg, Department for Computer Science, submitted September 2004.
9. Heiko Rölke. Modellierung und Implementation eines Multi-Agenten-Systems auf der Basis von Referenznetzen. Diplomarbeit, Universität Hamburg, 1999.

A behavior model for IEC 61499 function blocks.

Mohamed Khalgui, Xavier Rebeuf, Françoise Simonot-Lion

LORIA(UMR CNRS 7503) - INPL Campus Scientifique B.P. 239 54506

Vandoeuvre-lès-Nancy cedex. France.

[khalgui, rebeuf, simonot]@loria.fr

This paper deals with the design of critical control-command industrial applications. This kind of applications has to respect some real time constraints. A component-based approach called function blocks (IEC 61499 standard) is classically used to specify such software and its execution support. A function block is defined as an event trigger component. To validate temporal properties, we propose a state machine model compliant with this standard. Nevertheless, a function block cannot handle simultaneous input event occurrences. During the execution, some event occurrences may be lost leading to non predictable behavior. For critical systems, events loosing is not suitable. We propose to design offline scheduling of FBs execution in order to avoid simultaneous event occurrences. The scheduling construction relies on sufficient schedulability conditions that we propose. Finally, we verify the scheduling correctness using the state machine model.

Keywords. Function Blocks, IEC 61499, Timed Automata, Real Time, offline scheduling, composition.

1 Introduction

Nowadays, developing industrial control applications remains an expensive and complex activity. Indeed, such applications have to be certified with regard to several functional and extra-functional properties. Since they control critical processes, one of the most important property deals with Real Time behavior. Several approaches have been proposed to develop safe control applications. They allow to model applications at design time [2, 16, 19]. Nevertheless, it is difficult to evaluate real-time behavior without modeling the execution support.

The IEC 61499 standard [10] is one of the most known approaches allowing to design applications as well as the execution support. It is a component-based methodology [4]. According to such standard, a component (named Function Block) is a reusable functional unit of software that has its own data structure manipulated by one or more algorithms.

A function block interacts with its environment thanks to event and data inputs and outputs. Event inputs trigger the function block activation while data inputs provide algorithms parameters. The composition of function bocks consists in linking inputs and outputs via channels. Therefore, a control application is specified by a so called “function block network”. This functional specification can be distributed among various physical devices connected on a field bus network.

Several real applications have been specified using FBs [11]. Moreover, Function block libraries already exist [9]. The component-based approach allows modular verificaion avoiding combinatory explosion. Therefore, this approach is a good framework to validate a priori temporal properties.

Nevertheless, in the standard, a function block cannot handle simultaneous input event occurrences. It selects one occurrence and discards the others. During the execution, some event occurrences may be lost. This phenomena can lead to non predictable behavior. For critical systems, event loosing is not suitable [14].

In this paper, we propose to model applications built according to the standard using timed state machines [1]. Thanks to the UPPAAL model checker tool [13], it is possible to check and simulate an application on a resource and to verify temporal properties. Face to previous approaches [18, 20, 6], we model a function block as an autonomous component responsible for the input event management. Moreover, we take into account scheduler inside a resource model. This allows to validate not only a function block, but also a complete application executed on one resource.

To solve event occurrence loosing, we characterize the temporal application behavior. Based on this temporal specification, we propose to define sufficient schedulability conditions avoiding simultaneous occurrences. We construct safe offline scheduling policy for the resource. Finally, we propose to validate our scheduling policy using the state machines model.

In the next section 2, we briefly present the IEC 61499 standard. Then, we discuss some previous modeling approaches. The section 4 describes our model based on state machines. In the section 5, we present the schedulability conditions. Finally, we illustrate our approach thanks to a classical example.

2 The IEC 61499 concepts

We present the main concepts of the IEC 61499 Function Block standard. This standard is an extension of the IEC 61131.3 [12] for the Programmable Logic Controller. We can divide its description into two parts: the static description and the behavior one.

2.1 Static description

An application function block (FB) is a functional unit of software that supports some functionalities of an application (see Figure 1). It is composed by an interface and an implementation. The interface contains data/event inputs and outputs supporting the interaction with the environment. Events are responsible for the activation of the function block while data contain valued information. According to the standard, the association between an event input and a data input is modelled by a link connecting them. The implementation consists of a body and a head.

The body is composed of internal data and algorithms implementing the block functionalities. Each algorithm gets values in the input data channel and produces values in the output data channels. They are programmed in structured text (ST) language [12].

The block head is connected to event flows. It selects the sequence of algorithms to execute with regard to an input event occurrence. The selection mechanism of the event occurence is encoded in a state machine called Execution Control Chart (ECC). At the end of the algorithms execution, the ECC sends the corresponding output event occurrence.

In the standard, a function block network defines the functional architecture of a control application. Each function block event input (resp. output) is linked to an event output (resp. input) by a channel. Otherwise, it corresponds to a global application input (resp. output). Data inputs and outputs follow the same rules.

The execution support architecture (i.e. the industrial control system) is defined by a devices network (figure2). A device is composed of one processing unit,sensors, actuators

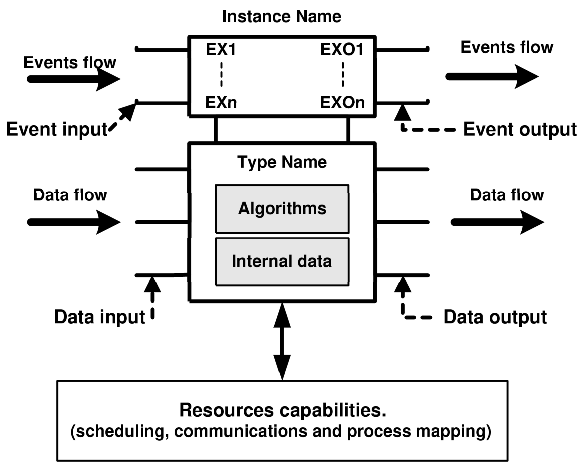


Figure 1: The Function Block concept.

and network interfaces. In order to manage the scheduling, a device contains several resources. A resource is a logic execution unit corresponding to a periodic and fixed time slot of the processing unit. It provides scheduler for a set of Function Blocks.

The operational architecture corresponds to a distribution of the application function blocks over the different ressources of the execution support architecture. Note that several applications can be distributed over the same execution support architecture[14]. The advantage of such approach is to take into account hardware and software components.

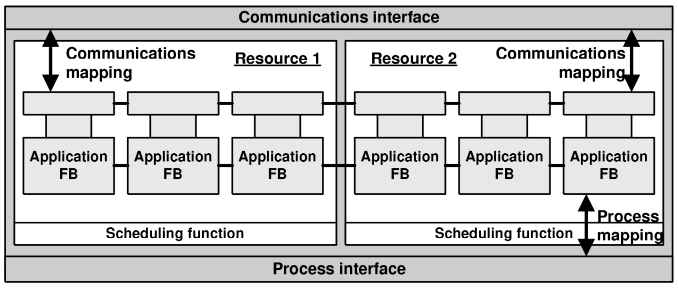


Figure 2: A device architecture.

2.2 Behavior description

Let turn to the behavior description of a function block. Note that only algorithms executions spend time. In a given function block, the ECC is said *idle* if there is no algorithm execution. Otherwise, the ECC is *busy*. As in synchronous programming [3], all events occurring when the ECC is busy are considered simultaneous. We can devise the ECC behavior into three steps:

- First, it selects one input event occurrence according to priority rules defined in the resource. Then, it clears all the event input buffers. Note that the other simultaneous event occurrences are lost.

- It activates the algorithm sequence corresponding to the selected event. Then, it waits for the resource scheduler to process the sequence.
- When the execution sequence ends, it emits the corresponding output event occurrence.

In the standard, the resource scheduler applies only non preemptive policies.

3 Previous modeling approaches

In order to validate the behavior of an operational architecture, several modeling approaches have been proposed.

The first one [20] is based on the NET Condition Event Systems as extension of the Petri Nets formalism [8]. To specify the FB behavior, five modules are proposed. For each FB event input, an Event Input State Machine (EISM) and an Event Input Variables Storage (EIVS) must be instantiated. The EISM module encodes the behavior of the event input while the EIVS stores the occurrence in a variable. An EC Operation State Machine (ECO-SM) is proposed to control the ECC module. A module for each event output variable stores the results. The modeling complexity of a function block (five modules) can lead to a combinatorial explosion when we validate real applications. Moreover, in this model, event selections are performed outside the Function Block. Therefore, Function Block components are not autonomous. Finally, the approach does not model the function block scheduling. Therefore, this work cannot validate temporal behavior of the whole application[19].

The second approach [6] proposes to model FB using a synchronous language called SIGNAL [3]. It uses clocks to assure the synchronization between the ECC and the input events in order to avoid occurrence loosing. Therefore, this approach is not fully compliant with the standard.

The last approach [17] uses the Timed Automata formalism[1]. It specifies a FB behavior thanks to two kind of modules: one state machine per each event input and one for the ECC. It models a resource with two modules : one ECC invocation automaton and one event input clear state machine per each Function Block. The authors propose resource scheduler accepting requests only when the processor is free. Nevertheless, the model does not specify a scheduling policy for managing the queue. Therefore, the scheduling can be non deterministic. Moreover, as for the first approach, the event input management is dedicated to the resource: FB components are not autonomous.

4 The Function Block Model

We propose a way for the implementation of a FB that is compliant to the standard IEC 61499 and based on a formal approach. For this purpose, we use the timed automata formalism. A FB is modeled with two kinds of modules: one state machine per each event input and one automaton for the ECC. In each block, an event input is characterized by a unique priority. Different event inputs have different priorities. Furthermore, each FB memorizes the highest priority of events occured from the last invocation of the ECC and discards the other ones.

Therefore, the use of an invocator as in [20, 18] is useless. Interactions between the event inputs and the ECC occur directly inside the FB. Note that our approach models the occurrence loosing.

We use the timed automata formalism of the UPPAAL tool [13]. For sake of conciseness, we only consider one device containing one resource. Therefore, temporal behavior of the application only depends on the FB network.

We first present the Function Block model. Then, we propose the resource model.

4.1 The Function block model

Let FB_jbe a function block composed by

- n event inputs denoted by EX_1, .., EX_n.
- m event outputs denoted by EXO_1, .., EXO_m. where $m \leq n$
- n algorithms denoted by $Alg_{(j,1)}$, ..., $Alg_{(j,n)}$.

In this paper, we take the "connections" FBs model as a model for our theoretical approach. Therefore, we suppose that for each input event EX_i, we associate one and only one correspondent algorithm $Alg_{(j,k)}$ and an output event EXO_e.

Since we want to validate temporal behavior, we only focus on the control flows defined by events occurrences. We do not consider data flows. Each algorithm Algi is classically characterized by:

- a minimum execution time $BCET(Alg_{(j,i)})$ (Best Case Execution Time).
- a maximum execution time $WCET(Alg_{(j,i)})$ (Worst Case Execution Time).

Note that these values correspond to the execution of the algorithm alone on the considered device.

4.1.1 The Event Input Model

We present the state machine modeling one Event input. Let $priority(EX_i)$ be the priority level of EX_i as defined for this Function Block FB_j. The higher level is 0 and the lower one is n-1. We use a variable termed a Waiting Highest Priority and denoted by WHP. This variable stores the highest priority of events occured from the last invocation of the ECC. At the beginning, WHP is initialized to n.

The Figure 3 displays the generic model of the event input EX_i.

When an event occurrs on EX_i, two cases exist:

1. if the WHP is lower than its priority level, then there exists (on another input) a current occurrence with a higher level priority. In this case, the occurrence on EX_i is discarded.
2. Otherwise, it has the highest priority. In this case:

- WHP is updated with its priority.
- the state machine waits the ECC.

When the ECC will be waiting for *intern*, the selected occurrence (on the highest priority input) is stored in the variable *selectAlg*(j) (where j is the FB identifier). This identifier is used by the ECC to identify algorithm to perform. Then the WHP is initialized to n to start the next selection process.

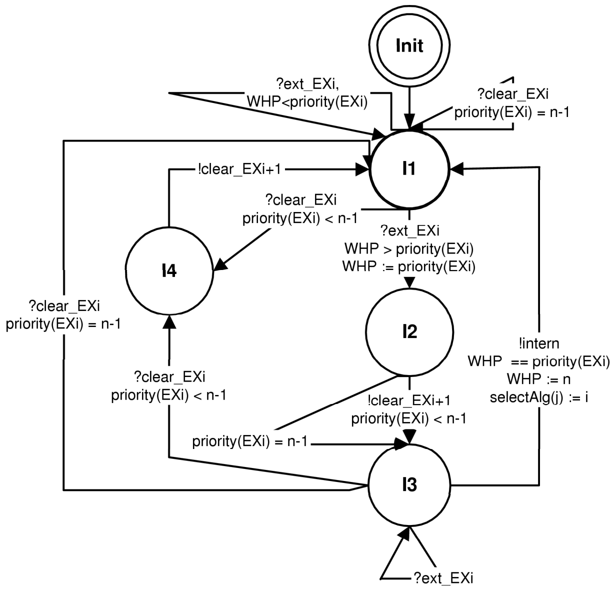


Figure 3: The generic model of an event input.

Note that the UPPAAL tool doesn't support the event broadcast. Therefore, we propose a solution based on recursive clear. If a state machine of an event input EX_i receives a clear event, it sends it to the automata of the input EX_{i+1}.

We can simplify this generic model into three sub models. The Figure 4 shows the models of:

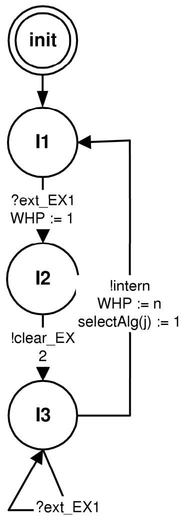
- the event input having the highest priority (a),
- the event input with the lowest priority (b),
- the rest of event inputs (with neither the highest nor the lowest priority) (c).

4.1.2 The ECC model.

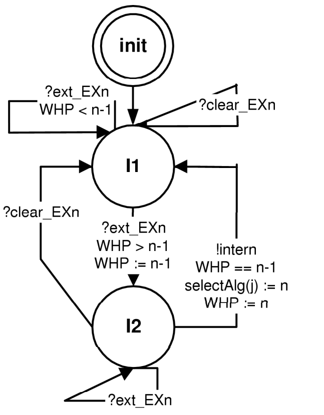
We extend the ECC model proposed in [17] (see Figure 5). For sake of clarity, we consider that the event input EX_i corresponds to the invocation of the algorithm $Alg_{(j,i)}$.

Let consider the function block FB_j. If the ECC is idle, then it waits for the *intern* event from one of the event input state machines. Once the *intern* event occurs (the occurrence input is selected), the ECC waits the resource scheduler. The latter authorizes the execution by sending the event *Exe_FB* when the variable *procFB* is equal to j.

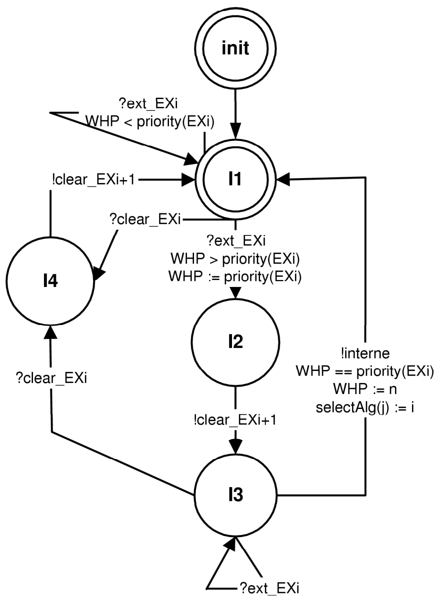
At the completion of the algorithm, the ECC receives the event *end_Exec* from the scheduler. Finally, the ECC sends an occurrence in the corresponding output event channel.



(a) The event input with the highest priority.



(b) The event input with the lowest priority.



(c) The event input (with neither the highest nor the lowest priority)

Figure 4: Event Input models.

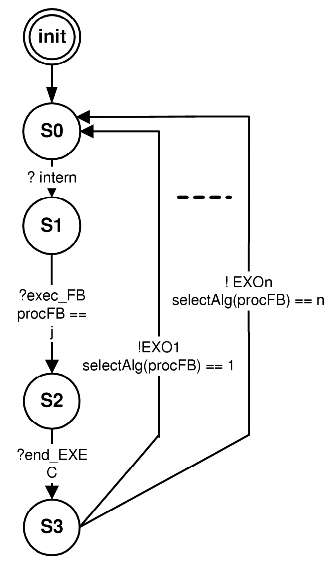


Figure 5: The ECC model.

4.2 The Resource Model

The resource behavior is controlled by a scheduler. This control can be modeled as a state machine. We consider a periodic offline scheduling stored in an array called *Sched*. We postpone the *Sched* construction to the section 5. The figure 6 displays the scheduler state machine.

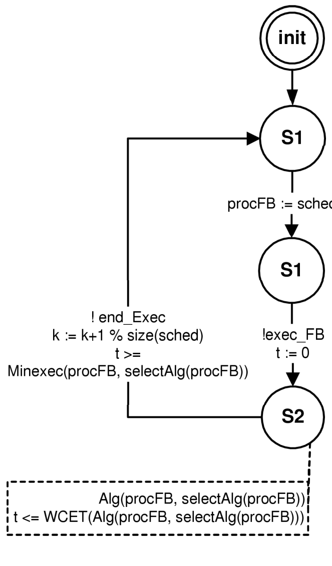

init
S1
procFB := sched(k)
S1
!exec_FB t := 0
S2
! end_Exec k := k+1 % size(sched) t >= Minexec(procFB, selectAlg(procFB))
Alg(procFB, selectAlg(procFB)) t <= WCET(Alg(procFB, selectAlg(procFB)))

Figure 6: The scheduler model.

When the scheduler is idle, then it authorizes the execution of the next Function block FB_j according to the *Sched* array. The execution time of the selected algorithm $Alg_{(j,i)}$ is encoded in the state S_2.

- If the scheduler implements idling policy, then the execution time is equal to $WCET(Alg_{(j,i)})$.

$$Minexec(j,i) = WCET(Alg_{(j,i)})$$

- Otherwise, it is bounded between $BCET(Alg_{(j,i)})$ and $WCET(Alg_{(j,i)})$.

$$Minexec(j,i) = BCET(Alg_{(j,i)})$$

The ECC signals the execution end with the event *end_exec* and it becomes idle.

5 Offline scheduling design

According to the IEC 61499 standard, the proposed model allows occurrences loosing [6]. This problem can be critical for some applications that becomes non deterministic. therefore, we propose a condition to construct non-preemptive offline scheduling [5, 7] avoiding simultaneous event occurrences for a given Function Block.

We first characterize the input events arriving law for a function block. Then, we propose a sufficient condition of schedulability to avoid occurrences loosing. Finally, if the condition is satisfied, we compute the emission law for the output events.

5.1 Characterization of input events

Let FB_j be a function block having n event inputs EX_1, EX_2, ..., EX_n. We classically define the input occurrences arriving law. We only consider periodic input events EX_i. Let denote by:

- $Of(EX_i)$ the offset of the event (it corresponds to the arrive time of the first occurrence),
- $P(EX_i)$ the period,
- $G(EX_i)$ the jitter (it corresponds to the maximum deviation of the period for the arrive time of an occurrence).

Let denote by $t(EX_{i,m})$ the arrive time of the occurrence m of the event EX_i. Such occurrence is denoted by $(EX_{i,m})$. Such time is bounded as follows:

$$\begin{gathered} Of(EX_i) + [m.P(EX_i)] \\ \leq \\ t(EX_{i,m}) \\ \leq \\ Of(EX_i) + [m.P(EX_i)] + G(EX_i); m \in \mathbb{N} \end{gathered}$$

5.2 Schedulability condition

This characterization of input events allows to define sufficient conditions to ensure that all the events are taken into account. An EX_i occurrence is taken into account if it has the highest priority between all current occurrences. Therefore, to accept all the occurrences, we have to avoid several simultaneous ones. Let define the minimum duration between two occurrences to ensure that they are not simultaneous.

We denote $WCET\_MAX(FB_j)$ the maximum algorithm worst case execution time for the function block FB_j.

$$WCET\_MAX(FB_j) = Max_{i \in [1,n]}\{WCET(Alg_{(j,i)})\}$$

We define the $AP(FB_j)$ as the biggest activation period of the ECC. It corresponds to the maximum period where the ECC is busy. Therefore, all the events arriving during this period can be considered by the ECC as simultaneous occurrences.

$AP(FB_j)$ is composed by the maximum waiting time for the ECC to obtain the scheduler $PW(FB_j)$ and the worst case execution time of the function block ($WCET\_MAX(FB_j)$).

$$AP(FB_j) = PW(FB_j) + WCET\_MAX(FB_j)$$

Note that the waiting time of the scheduler ($PW(FB_j)$) depends on the scheduling policy. The figure 7 presents an execution sequence corresponding to a given scheduling policy. For example, the period "events selection 3" corresponds to the maximum duration where arriving occurrences can involve in the selection for the execution windows 3. Therefore, they can be considered as simultaneous.

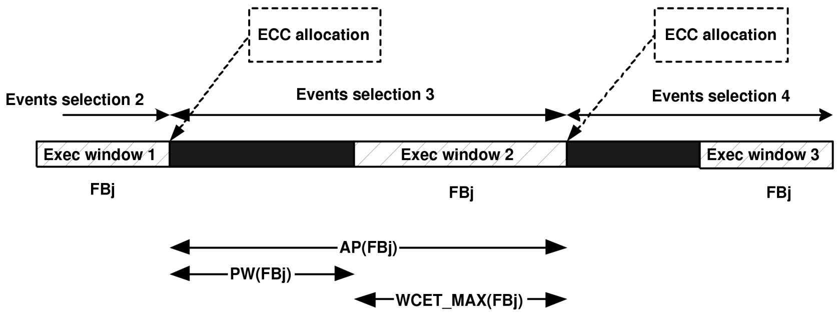


Figure 7: An execution scenario in a resource.

To ensure that there is no simultaneous occurrences, we propose a sufficient condition on the scheduling policy.

Theorem 1 ***(schedulability condition).*** *The scheduling policy avoids loosing occurrences for the Function Block FB if it respects the condition:*
$\forall i = 1...n, \forall j = 1...n, \forall k, m \in \mathbb{N},$ *such as* $(EX_{i,k}) \neq (EX_{j,m})$

$$\begin{gathered} PW(FB) \\ < \\ \mid Of(EX_i) + (k.P(EX_i)) - Of(EX_j) - (m.P(EX_j)) - G(EX_j) \mid \\ -WCET\_MAX(FB) \end{gathered}$$

Proof. *To avoid the loosing phenomena, the duration between two different event occurrences must be greater than $AP(FB)$. If we consider two periodic events, the minimum interval occurs when the first EX_j occurrence arrives late (i.e. with the jitter) and the second EX_i occurrence arrives in time (i.e. without any jitter). We obtain the following formula:*

$\forall i = 1...n, \forall j = 1...n, \forall k, m \in \mathbb{N},$ *such as* $(EX_{i,k}) \neq (EX_{j,m})$

$$\begin{gathered} \mid Of(EX_i) + (k.P(EX_i)) - Of(EX_j) - (m.P(EX_j)) - G(EX_j) \mid \\ > \\ AP(FB) \end{gathered}$$

where

$$AP(FB) = PW(FB) + WCET\_MAX(FB)$$

Therefore, we obtain the theorem condition.

Remarks.

- In particular, if $i = j$ and $m = k - 1$, two successive occurrences have to respect the condition:

$$\mid P(EX_i) - G(EX_i) \mid > AP(FB).$$

- This theorem presents sufficient condition. Since we only consider maximum times, if two occurrences don't respect the condition, then they can not be considered simultaneous in some executions.

5.3 Characterization of output events

In order to be able to compose the function blocks, we have to characterize the output events that can be input events for the next FB. such characterization depends on the scheduling policy.

Let consider an input occurrence $(EX_{i,n})$ of a function block FB_j. We want to compute the time $t(EXO_{i,n})$ of the corresponding output occurrence $EXO_{i,n}$. We can compute a lower bound and an upper one (see figure 8).

The lower bound occurs when the following conditions are satisfied:

- $(EX_{i,n})$ arrives with no jitter: $tmin(EX_{i,n}) = Of(EX_i) + [n.P(EX_i)]$
- $(EX_{i,n})$ arrives when the ECC is idle.
- $(EX_{i,n})$ arrives when an execution window for FB starts.
- The $Alg_{(j,i)}$ execution time corresponds to the minimum execution time $Minexec(j, i)$.

The upper bound occurs when the following condition are satisfied:

- $(EX_{i,n})$ arrives with a jitter: $tmax(EX_{i,n}) = Of(EX_i) + [n.P(EX_i)] + G(EX_i)$
- $(EX_{i,n})$ arrives when the ECC is busy.
- $(EX_{i,n})$ arrives at the beginning of the selection period for the execution window 2.
- The $Alg_{(j,i)}$ execution time corresponds to the maximum execution time $WCET(Alg_{(j,i)})$.

Therefore, we obtain the following formula :

$$\begin{gathered} Of(EX_i) + [n.P(EX_i)] + Minexec(j, i) \\ \leq \\ t(EXO_{i,n}) \\ \leq \\ Of(EX_i) + [n.P(EX_i)] + G(EX_i) \\ +2PW(FB) + WCET\_MAX(FB_j) + WCET(Alg_{(j,i)}) \end{gathered}$$

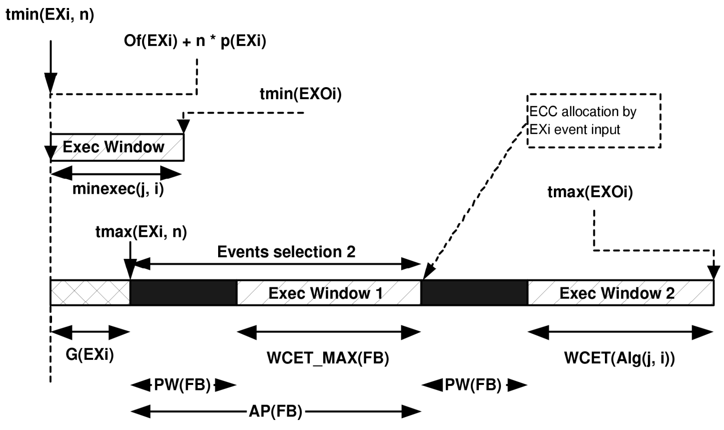


Figure 8: A temporal characterization of an output event.

The output event EXOi has the following characteristics:

- $Of(EXO_i) = \quad Of(EX_i) + Minexec(j, i)$
- $P(EXO_i) = \quad P(EX_i)$
- $$\begin{aligned} G(EXO_i) = \quad & G(EX_i) + 2PW(FB) + WCET\_MAX(FB) \\ & +(WCET(Alg_{(j,i)}) - Minexec(j, i)) \end{aligned}$$

To be considered as a periodic event, the EXO_i jitter has to be lower than the period $(G(EXO_i) < P(EX_i))$. Therefore, we can deduce an additional condition of the scheduling policy.

Theorem 2 ***(schedulability condition 2).*** *To obtain a periodic output event, the scheduling policy has to respect the condition:*
$\forall Exi$ *input periodic event of FB.*

$$\begin{gathered} PW(FB) \\ < \\ (1/2).(P(EX_i) - G(EX_i) - WCET\_MAX(FB_j) \\ -WCET(Alg_{(j,i)}) + Minexec(j, i) \end{gathered}$$

Proof. *The proof is straightforward.*
We classically distinguish two policy types:

- *Non idling policy:* the scheduler signals the end of algorithm execution as soon as it occurs. In this case, we have:

$$Minexec(j, i) = BCET(Alg_{(j,i)}).$$

- *Idling policy:* the scheduler waits during WCET of the algorithm before signaling the end of the execution. In this case, we have:

$$Minexec(j, i) = WCET(Alg_{(j,i)}).$$

6 Example.

To illustrate the approach, we present the classical saw tooth application. A full description of this example can be found in [14] . The Saw tooth is an application specified by a network of FBs (figure 9). We first describe the application. Then we characterize the temporal behavior and we propose an idling and non-idling policy.

6.1 Application description

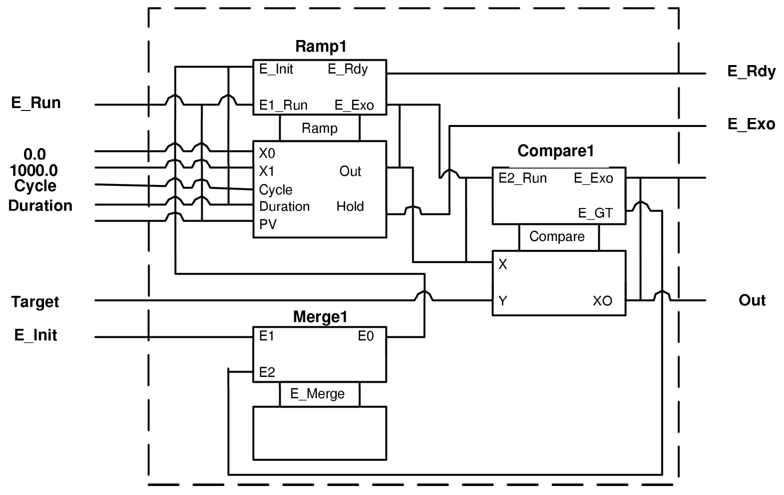


Figure 9: The sawtooth application design.

The application goal is to provide an output value “Out” that follows a ’sawtooth’ profile. In fact, it repeatedly increases the output from 0.0 to a ’Target’ value and then resets and restarts the ramp again from 0.0.

The application is initialized using an initialization event E_init. When E_init occurs, the input values for X0, X1, Cycle and Duration are stored within the ’Ramp1’ input variables. In the example, X0 and X1 are constants (X0=0, X1=1000) and bound the Ramp output variable ’Out’. The values of Cycle and Duration come from values provided at the external interface of the application. They define the timing characteristics of the ’saw tooth’ wave form. The Ramp Function block is composed of two algorithms:

- Alg_init corresponding to the Ramp initialization
- ALG_E1_RUN invoked by the event E1_RUN.

After initialization, the application is ready to receive a regular stream of events at the event input E1_Run. Each E1_Run event occurrence triggers the Ramp1 block to increment its output ’Out’ towards the upper limit set by X1. Each time the Ramp1 internal algorithm completes its execution, an output event E_Exo is passed to the Compare block. It checks the value of ’Out’ (stored in X) in the block against the ’Target’ value (stored in Y). If the X is lower than Y, then the compare block outputs the value of X to output XO and issues event E_Exo. Otherwise, the compare block triggers its output event E_GT. This event is fed back to the Merge1 block which produces a new initialization event for the Ramp1 block. The Ramp1 is then re-initialized and the output is reset to 0.0.

The slope of the saw tooth profile can be changed by re-initializing the block with an initialization event at event input E_init issued with a different Duration value. Merge1 is an instance of one of the standard event function blocks and provides a means for multiple events to be merged into a new stream of events. In this case, Merge is used to allow the Ramp block to be initialized either by an external event on the event input E_init or from an internally generated event coming from the compare block. We suppose that it doesn't consume processor time.

In the example, we are interested to characterize the application behavior in the order to guarantee that each E1_Run event is taken into account. We suppose also that the input event E_Init is not periodic but it has the higher priority than the input event E1_Run. Therefore, we only consider the execution between two E_init occurrences.

6.2 initial temporal characterization

Let E1_Run be a periodic event with the following characteristics.

- Of(E1_Run) = 5 (relatively to the last E_Init occurrence).
- P(E1_Run) = 40.
- G(E1_Run) = 2.

For the Function Block Ramp1, we define:

- WCET(Alg_E1_Run) = 4.
- BCET(Alg_E1_Run) = 2.
- WCET(Alg_E_init) = 4.
- WCET_MAX(Ramp) = 4.

For the Function Block Compare1, we define:

- WCET(Alg_E2_Run) = 2.
- BCET(Alg_E2_Run) = 0.
- WCET_MAX(Compare) = 2.

6.3 Non-idling scheduling policy

To have a correct behavior of the Ramp FB, we must respect the two scheduling conditions (theorem1 and 2) where Minexec=BCET.

$$PW(Ramp) <\mid P(E1\_Run) - G(E1\_Run) \mid - WCET\_MAX(Ramp)$$

$$PW(Ramp) < (1/2).(P(E1\_Run) - G(E1\_Run) - WCET\_MAX(Ramp) - WCET(Alg\_E1\_Run) + BCET(Alg\_E1\_Run))$$

In result, we have as temporal condition:

$$\boxed{PW(Ramp) < 16}$$

We compute the event output E_Exo is characteristics.

- Of(E_Exo) = 7.
- P(E_Exo) = 40.
- G(E_Exo) = 8 + 2 * PW(Ramp).

Note that the E_Exo characteristics correspond to the E2_Run ones. To have a correct behavior of the Compare FB, We must respect the two scheduling conditions

$$PW(Compare) <\mid P(E2\_Run) - G(E2\_Run) \mid - WCET\_MAX(Compare)$$

$$PW(Compare) < (1/2).(P(E2\_Run) - G(E2\_Run) - WCET\_MAX(Compare) - WCET(Alg\_E2\_Run) + BCET(Alg\_E2\_Run))$$

Instantiating the variables, we obtain:

$$\boxed{PW(Compare) <\mid 32 - 2 * PW(Ramp) \mid -2}$$
$$\boxed{PW(Ramp) + PW(Compare) < 14}$$

Let consider the cyclic scheduling:

- Sched(0)=Ramp
- Sched(1)=Compare

In this case, we have to prove that the execution of Ramp (resp. Compare) can be included during the Waiting time (PW) of Compare (resp. Ramp):

$$\boxed{WCET\_MAX(Ramp) \leq PW(Compare)}$$
$$\boxed{WCET\_MAX(compare) \leq PW(Ramp)}$$

Since these 5 inequations have a solution, the scheduling avoids events loosing.

6.4 Idling scheduling policy

Applying the same method as described in the previous section, we obtain the following inequations for Idling scheduling policy (i.e. Minexec=WCET).

$$\boxed{PW(Ramp) < 17}$$

$$\boxed{PW(Compare) <\mid 34 - 2 * PW(Ramp) \mid -4}$$
$$\boxed{PW(Ramp) + PW(Compare) < 17}$$

Let consider the same cyclic scheduling:

- Sched(0)=Ramp
- Sched(1)=Compare

In this case, we have to prove that the execution of Ramp (resp. Compare) can be included during the Waiting time (PW) of Compare (resp. Ramp):

$$\boxed{WCET\_MAX(Ramp) \leq PW(Compare)}$$
$$\boxed{WCET\_MAX(compare) \leq PW(Ramp)}$$

Since these 5 inequations have a solution, the scheduling avoids events loosing.

Using the model described in section 4 and the UPPAAL tool, we verify that the application scheduling avoids the loosing phenomena and lets to a correct behavior.

7 Conclusion and future works

This paper presents contributions to develop industrial control applications. It is based on the IEC 61499 defining a modular approach for application and execution support design. We propose a state machine model to validate temporal properties. It takes into account the functional specification and its distribution over the execution support. Moreover, we define a resource scheduler able to take into account offline scheduling policy.

Once defining the scheduling policy, such model allows to verify that an execution is free of simultaneous events. Nevertheless, it cannot help for designing a safe scheduling policy. Therefore we propose two schedulability conditions based on the temporal characterization of the application. On the saw tooth example, we construct offline scheduling policies (idling and non-idling) compliant with these conditions.

We are currently working on a method to automatically generate safe offline scheduling policy. We also develop an on-line scheduler [16] able to dynamically evaluate and handle deadline for each FB execution. We extend our model in order to take into account several resources on one device. Then we plan to consider parallel execution distributed on devices network.

In future works, we will study decomposition of temporal property in order to perform modular validation of a function blocks application. The goal is to replace a FB by an equivalent one (regarding to the property) without checking the whole application.

References

[1] R. Alur, D. Dill. *A Theory of Timed Automata.* 17th International Colloquium on Automata. 1990.

[2] Articus. *Rubus OS Reference Manual.* Articus Systems. 1996.

[3] A. Benveniste, B. Le Goff, P. Le Guernic. *Hybrid dynamical systems theory and the language "SIGNAL".* INRIA RR-0838. France.

[4] I. Crnkovic, M. Larsson. *Building reliable component-based software systems.* Artech House. London. ISBN 1- 58053-327-2

[5] C. Dima, A. Girault, C. Lavarenne, Y. Sorel. *Off-line real-time fault-tolerant scheduling.* Euromicro Workshop on Parallel and Distributed Processing. Italy. 2001.

[6] J.M. Faure, C. Schnackenbourg, J.J. Lesage. *Toward IEC 61499 Function Blocks diagrams verification.* IEEE International Conference on Systems Man and Cybernetics. SMC'2002. Tunisia. 2002.

[7] L. George, P. Muhlethaler, N. Rivierre. *Optimality and non-preemptive real-time scheduling revisited.* RR-2516. 1995. INRIA. France.

[8] S. Haddad, D. Poitrenaud. *Checking Linear Temporal Formulas on Sequential Recursive Petri Nets.* 8th International Symposium on Temporal Representation and Reasonning (TIME-01). Italy. 2001.

[9] www.holobloc.com

[10] IEC TC65 WG6. *Function Blocks for Industrial Process Measurements and Control Systems.* Committee Draft. 2003.

[11] http://www.ifak-md.de/wg7/

[12] International Standard IEC 1131-3. *Programmable Controllers Part 3. Bureau Central de la commission Electrotechnique Internationale.* Switzerland. 1993.

[13] G. Behrmann, A. David, K. G. Larsen, O. Möller, P. Pettersson, W. Yi. *UPPAAL - Present and Future.* The 40th IEEE Conference on Decision and Control (CDC'2001). USA. 2001.

[14] R. Lewis. *Modelling Control systems using IEC 61499.* The Institution Of Electrical Engineers. ISBN 0 85296 796 9.

[15] www.pecos-project.org

[16] W-K. Shih, J. W. S. Liu. *On-Line Scheduling of Imprecise Computations to Minimize Error.* SIAM journal of computing. Volume 25. number 5. pp 1105-1121. Society for Industrial and Applied Mathematics. 1996.

[17] M. Stanica, H. Guéguen. *A Timed Automata Model of IEC 61499 Basic Function Blocks Semantic.* ECRTS'03, Euromicro European Conference on Real-Time Systems. Portugal. 2003.

[18] D. B. Stewart, R. A Volpe. P. K. Khosla. *Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects.* IEEE Transactions on Software Engineering. 1997.

[19] V. Vyatkin. H-M Hanisch. *Modeling Of IEC 61499 Function Blocks A Clue To Their Verification.* XI Workshop on Supervising and Diagnostics of Machining Systems. Wroclaw. 2000.

[20] V. Vyatkin. H-M. Hanisch. *A modeling approach for verification of IEC1499 function blocks using Net Condition/Event Systems.* IEEE conference on Emerging Technologies in Factory Automation (ETFA'99). Spain. 1999.

Compositional Analysis of Mobile Agents using Structural Invariants of Object Nets

Michael Köhler, Heiko Rölke, and Rüdiger Valk

University of Hamburg, Department of Informatics
Vogt-Kölln-Str. 30, D-22527 Hamburg
Phone: +49 40 42883-2407 Fax: +49 40 42883-2246
{koehler,roelke,valk}@informatik.uni-hamburg.de

Abstract. Mobility induces new challenges for dynamic systems, which need a new conceptional treatment: systems, that deal for example with mobile agents, need extended security concepts to handle the risks, induced by foreign, untrusted agents.
In this work we apply structural analysis methods for object nets – namely place invariants – to a case study modelling mobile agents.

Keywords: agent system, compositionality, mobility, nets as tokens, object nets, security

1 Introduction

The general context of this presentation is mobility in open systems, with a special emphasis on multi-agent systems to illustrate the underlying concepts. In this context the question of security issues arises, especially the questions of how platforms can be protected against malicious agents.[1] As discussed in [Ch98], security issues, which arise due to the mobility of agents, are conceptually different from those in traditional systems.

So, a natural questions arise in this context: *"How can integrity of platforms be achieved without having knowledge about the agents in advance?"* There are two approaches to establish the platform security (cf. [Ts99]). The first approach demands a proof of the agent, that it is harmless. This approach is known as "proof-carrying code" (cf. [NL97]). The second approach is to encapsulate each agent in an environment – called "sandbox" – which enforces harmless processes. A prominent example is the security mechanism in Java (cf. [Sun97]).

Here, we propose to use object nets as a formalism for mobile agents. Object nets are well suited to express mobility (cf. [KMR03]). Beside the modelling aspect object nets provide the possibility of *structural* analysis by using linear invariants. The invariant calculus for object nets has the property of compositionality, i.e. structural properties of platforms can be proven independently from those of the mobile agents. This implies the key issue that structural analysis can used (a) to prove that an agent is harmless and (b) to prove that a platform enforces harmless processes for each possible agent moving onto it.

Currently the concept of mobile entities is an research area of actual interest. Most of the work on mobility is based on mobility calculi, like the ambient-

[1] For the opposite question, how agents could be protected against untrusted platforms, [ST98] formulates a principle solutions based on cryptographic methods.

[CGG99], seal [VC98] or π-calculus [MPW92]. For this area verification relies on the notion of bisimulation rather than invariant analysis. Practical approaches in form of graphical modelling languages like AUML (Agent Unified Modelling Language) (cf. [BOP00]) are not equipped with an exact formalism.

Petri nets provide both a graphical and a formal representation of systems in the notion of the token game. For the application area of mobility the formalism of Mobile Object Net Systems [KR03] (based on the *nets-within-nets* paradigm of [Va98]) provides a suitable conceptual framework, since nets are allowed as tokens. The paradigm of nets-within-nets has been adopted by other approaches, for example the formalism of Nested Petri Nets [Lo00]. Its relationship to linear logic has been discussed by [Fa00]. Another example for the use of Petri nets in the context of mobility is the formalism of Mobile Petri Nets [Bu99], which provides a description of mobility by an embedding of the π-calculus. For an overview of the relationship of Petri nets and mobility cf. [KMR03,KR04].

The paper has the following structure: Section 2 gives a short introduction into the paradigm of nets-within-nets. The formal definition of the semantics of mobile object net systems is given. In Section 3 we give a description how nets-within-nets are used in the concrete context of mobility in open systems. Mobile agents and their environment can be regarded as a two-level hierarchy of nets. In Section 4 a case study is analysed using structural approaches.

2 Object Systems

The simplest model of "nets-within-nets" are "Unary Elementary Object Systems" (Eos for short) defined by [Va98]. They are called "elementary" since the nesting hierarchy is limited to a depth of two. On the top level there is one so called *system-net* which has instance of one – therefore the term unary – *object-net*. Here, we give a generalised version of Eos, since Place/ Transition nets (short: P/T-nets) are considered (instead of EN systems as in [Va98]).

In Figure 1 a firing sequence is illustrated. The firing of t_1 created two net tokens, which are copies of the original one. The marking of the net token on p_1 is distributed to the copies. The distribution chosen in this sequence allows the two synchronisations (t_2, t_{11}) and (t_3, t_{12}). The effect of firing (t_2, t_{11}) modifies the marking of the net token on place s_2, but not the copy on s_3 – similarly for (t_3, t_{12}). Since the tokens on s_{13} and s_{14} are in different net tokens, transitions t_{13} is *not* enabled. The two net copies have to be recombined by the system-net transition t_4, resulting in a net token with the marking $s_{13} + s_{14}$.

It can be concluded, that object systems are very intuitive when dealing with mobile objects in a distributed system, since each place denotes a location for the net tokens independent from all other locations.

In the following we use the notation $\mathbf{x}$ for elements of the whole object-net system, $\hat{x}$ for elements of the object-net, and x for elements of the system-net.

Definition 1. *An* object system *is a tuple $OS = (N, \hat{N}, \rho)$, such that:*

– *The* system-net $N = (P, T, pre, post, M_0)$ *is a P/T-net with* $|{}^\bullet t|, |t^\bullet| > 0$ *for all* $t \in T$.

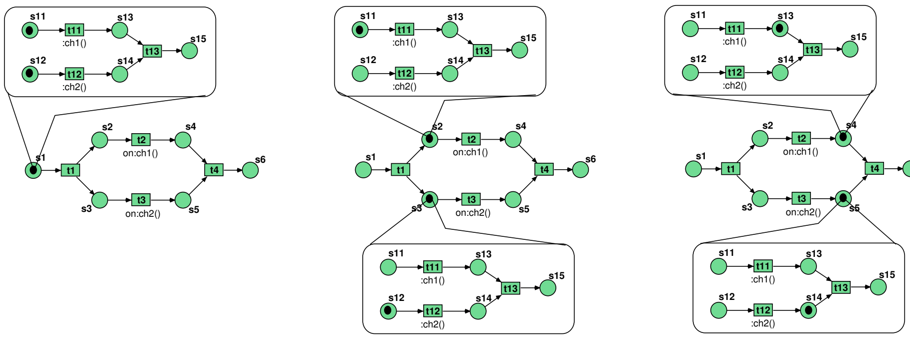


Fig. 1. Firing Sequence

– *The* object-net $\hat{N} = (\hat{P}, \hat{T}, \widehat{pre}, \widehat{post}, \hat{M}_0)$ *is a P/T-net disjoint from the system-net:* $(P \cup T) \cap (\hat{P} \cup \hat{T}) = \emptyset$.
– $\rho \subseteq T \times \hat{T}$ *is the* interaction relation.

The set of synchronising transitions in the system-net is $T_\rho := (\cdot\rho\hat{T})$. The set of synchronisation free transitions is $T_{\bar{\rho}} := T \setminus T_\rho$. Analogously, the set of synchronising transitions in the object-net is $\hat{T}_\rho := (T\rho\cdot)$. The set of synchronisation free transitions is $\hat{T}_{\bar{\rho}} := \hat{T} \setminus \hat{T}_\rho$. Thus, the set of transitions is $\mathbf{T} = (T_{\bar{\rho}} \cup \hat{T}_{\bar{\rho}} \cup \rho)$.

Markings are described by nested multisets. For Eos the nesting level is fixed with one level of nesting.

Definition 2. *A* marking *of OS is a multiset* $\mathbf{M} \in \mathcal{M}_v := MS(P \times MS(\hat{P}))$. *The* initial marking *of an* Eos *OS is* $\mathbf{M}_0((p, \hat{M})) = M_0(p)$ *if* $\hat{M} = \hat{M}_0$ *and* 0 *otherwise.*

By using projections on the first or last component of an Eos marking $\mathbf{M}$, it is possible to compare object system markings. The projection $\Pi^1(\mathbf{M})$ on the first component abstracts away the substructure of a net token, while the projection $\Pi^2(\mathbf{M})$ on the second component can be used as the abstract marking of the net tokens without considering their local distribution within the system-net.

Definition 3. *Let* $\mathbf{M} = \sum_{i=1}^{n}(p_i, \hat{M}_i)$ *be a marking of an* Eos $OS = (N, \hat{N}, \rho)$.

1. *Abstraction from the token substructure:* $\Pi^1(\sum_{i=1}^{n}(p_i, \hat{M}_i)) = \sum_{i=1}^{n} p_i$
2. *Summation of the distributed markings:* $\Pi^2(\sum_{i=1}^{n}(p_i, \hat{M}_i)) = \sum_{i=1}^{n} \hat{M}_i$

The Eos firing rule is defined for three cases: system-autonomous firing, object-autonomous firing, and synchronised firing. A transition can only occur autonomously if there exists no synchronisation partner in ρ, i.e. if $\tau \in (T_{\bar{\rho}} \cup \hat{T}_{\bar{\rho}})$. Otherwise if $\tau \in \rho$, only synchronous firing is possible.

The autonomous firing of a system-net transition t removes net tokens in the pre-conditions together with their individual internal markings. Since the markings of Eos are higher-order multisets, we have to consider terms $\mathbf{PRE} \in \mathcal{M}_v$ that correspond to the pre-set of t in their first component: $\Pi^1(\mathbf{PRE}) =$

pre(t). In turn, a multiset $\mathbf{POST} \in \mathcal{M}_v$ is produced, that corresponds with the post-set of t in its first component. Thus, the successor marking is $\mathbf{M}' = \mathbf{M} - \mathbf{PRE} + \mathbf{POST}$, in analogy to the successor marking $M' = M - \text{pre}(t) + \text{post}(t)$ of P/T-nets. The firing of t must also obey the *object marking distribution condition* $\Pi^2(\mathbf{PRE}) = \Pi^2(\mathbf{POST})$, ensuring that the sum of markings in the copies of a net token is preserved.[2]

An object-net transition $\hat{t}$ is enabled autonomously if in $\mathbf{M}$ there is an addend $(p, \hat{M})$ in the sum and $\hat{M}$ enables $\hat{t}$. For synchronous firing, a combination of both is required.

Definition 4. *Let $OS = (N, \hat{N}, \rho)$ be an* Ueos. *Let $\mathbf{M}, \mathbf{M}' \in \mathcal{M}_v$ be markings wrt. value semantics of OS. Then $\mathbf{M} \xrightarrow[OS]{\tau} \mathbf{M}'$ iff there exists $\mathbf{PRE}, \mathbf{POST} \in \mathcal{M}_v$ such that $\mathbf{PRE} \le \mathbf{M}$, $\mathbf{M}' = \mathbf{M} - \mathbf{PRE} + \mathbf{POST}$, and the following holds:*

$$\begin{aligned}
\tau = t \in T_{\bar{\rho}} \Rightarrow\ & \Pi^1(\mathbf{PRE}) = pre(t) \wedge \Pi^1(\mathbf{POST}) = post(t) \wedge \\
& \Pi^2(\mathbf{POST}) = \Pi^2(\mathbf{PRE}) \wedge \\
\tau = (t, \hat{t}) \in \rho \Rightarrow\ & \Pi^1(\mathbf{PRE}) = pre(t) \wedge \Pi^1(\mathbf{POST}) = post(t) \wedge \\
& \Pi^2(\mathbf{PRE}) \ge \widehat{pre}(\hat{t}) \wedge \\
& \Pi^2(\mathbf{POST}) = \Pi^2(\mathbf{PRE}) - \widehat{pre}(\hat{t}) + \widehat{post}(\hat{t}) \wedge \\
\tau = \hat{t} \in \hat{T}_{\bar{\rho}} \Rightarrow\ & \exists p : \Pi^1(\mathbf{PRE}) = \Pi^1(\mathbf{POST}) = p \wedge \\
& \Pi^2(\mathbf{PRE}) \ge \widehat{pre}(\hat{t}) \wedge \\
& \Pi^2(\mathbf{POST}) = \Pi^2(\mathbf{PRE}) - \widehat{pre}(\hat{t}) + \widehat{post}(\hat{t})
\end{aligned}$$

The following properties characterise the symmetries in the behaviour of an Eos.[3]

Proposition 1. *Let OS be an* Eos *as in Def. 1.*

1. *The abstract behaviour – determined by the projection Π^1 – on the system level of an* Eos *corresponds to the behaviour of the system-net viewed as a P/T-net: $\mathbf{M} \xrightarrow[OS]{t} \mathbf{M}'$ implies $\Pi^1(\mathbf{M}) \xrightarrow[N]{t} \Pi^1(\mathbf{M}')$, and $\Pi^1(\mathbf{M}')$ is uniquely determined.*
2. *The distributed object-net marking is invariant under autonomous actions of the system-net: $\mathbf{M} \xrightarrow[OS]{t} \mathbf{M}'$ implies $\Pi^2(\mathbf{M}) = \Pi^2(\mathbf{M}')$.*
3. *The behaviour of the distributed object-net is determined by Π^2 and is a sub-behaviour of the object-net viewed as a P/T-net: $\mathbf{M} \xrightarrow[OS]{\hat{t}} \mathbf{M}'$ implies $\Pi^2(\mathbf{M}) \xrightarrow[\hat{N}]{\hat{t}} \Pi^2(\mathbf{M}')$, and $\Pi^2(\mathbf{M}')$ is uniquely determined.*
4. *The abstract state of the system-net is invariant under object-autonomous actions: $\mathbf{M} \xrightarrow[OS]{\hat{t}} \mathbf{M}'$ implies $\Pi^1(\mathbf{M}) = \Pi^1(\mathbf{M}')$.*

[2] This represents the main difference of Eos compared with Valk's object systems [Va98], which require that each net token in **POST** hold all tokens $\Pi^2(\mathbf{PRE})$ of the net tokens from the pre-set. This kind of multiplication of system resources is inhibited in Eos. Note, that almost all propositions exploit this symmetry.

[3] Note, that not all properties hold in he semantics presented in [Va98], Since each net token in **POST** holds all tokens $\Pi^2(\mathbf{PRE})$ of the net tokens from the pre-set (i.e. tokens are duplicated) the distributed object-net marking $\Pi^2(\mathbf{M})$ cannot be preserved. So, Prop. 1 (2) and (5) become invalid for the semantics presented in [Va98].

5. *Synchronisation is an action composed of two sub-actions:* $\mathbf{M} \xrightarrow[OS]{(t,\hat{t})} \mathbf{M}'$ *implies* $\Pi^1(\mathbf{M}) \xrightarrow[N]{t} \Pi^1(\mathbf{M}')$ *as well as* $\Pi^2(\mathbf{M}) \xrightarrow[\hat{N}]{\hat{t}} \Pi^2(\mathbf{M}')$. *Furthermore,* $\Pi^1(\mathbf{M}')$ *and* $\Pi^2(\mathbf{M}')$ *are uniquely determined.*

Proof. Immediate from Definition 4. □

The compositionality of invariants is interesting for Eos. In the following we investigate how the invariant calculus of P/T-nets extends for Eos. Let $N = (P, T, \text{pre}, \text{post}, M_0)$ be a P/T-net. The incidence matrix Δ is defined by $\Delta(p,t) := \text{post}(t)(p) - pre(t)(p)$. A P-invariant $\boldsymbol{i} \in \mathbb{Z}^{|P|}, \boldsymbol{i} \neq \mathbf{0}$ is a vector that fulfils $\boldsymbol{i}' \cdot \Delta = \mathbf{0}$. Then every reachable marking $\boldsymbol{M}$ fulfils the linear equation $\boldsymbol{i} \cdot \boldsymbol{M} = \boldsymbol{i} \cdot \boldsymbol{M}_0$.

The following theorem states that invariants for a Eos can easily be composed from the invariants of the components.

Proposition 2. *Let* $OS = (N, \hat{N}, \rho)$ *be an* Eos *as in Def. 1 and let* $\boldsymbol{i} \in \mathbb{Z}^{|P|}$ *be an invariant of the system-net and* $\hat{\boldsymbol{i}} \in \mathbb{Z}^{|\hat{P}|}$ *one of the object-net. Then for all reachable marking* $\mathbf{M} \in \mathcal{M}_v$ *it holds:*

$$\boldsymbol{i} \cdot \Pi^1(\mathbf{M}) = \boldsymbol{i} \cdot \Pi^1(\mathbf{M}_0) \qquad \text{and} \qquad \hat{\boldsymbol{i}} \cdot \Pi^2(\mathbf{M}) = \hat{\boldsymbol{i}} \cdot \Pi^2(\mathbf{M}_0)$$

Proof. For a system-autonomous step $\mathbf{M} \xrightarrow{\tau} \mathbf{M}'$ we have $\Pi^1(\mathbf{M}) \xrightarrow[N]{\tau} \Pi^1(\mathbf{M}')$ by Prop. 1(1). Since $\boldsymbol{i}$ is an invariant of N we have $\boldsymbol{i} \cdot \Pi^1(\mathbf{M}) = \boldsymbol{i} \cdot \Pi^1(\mathbf{M}')$. Also, we have $\hat{\boldsymbol{i}} \cdot \Pi^2(\mathbf{M}) = \hat{\boldsymbol{i}} \cdot \Pi^2(\mathbf{M}_0)$, since $\Pi^2(\mathbf{M}) = \Pi^2(\mathbf{M}')$ by Prop. 1(2). Analogously with Prop. 1(3) and (4) for object-autonomous and steps and with Prop. 1(5) for synchronisations. □

3 Types of Mobility

It is quite natural to use object nets to model mobility and mobile agents. Each place of the system net describes a location that hosts agents, which are net tokens. Mobility can be modelled by moving the net token from one place to another. This hierarchy forms a useful abstraction of the system: on a high level the agent system and on a lower level of the hierarchy the agent itself.

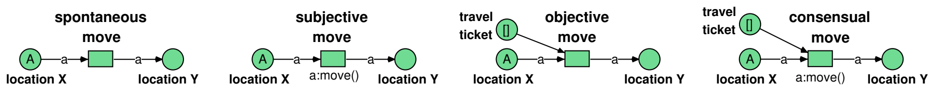


Fig. 2. Types of mobility

Without the viewpoint of nets as tokens, the modeller would have to encode the agent differently, e.g. as a data-type. This has the disadvantage, that the inner actions cannot be modelled directly, so, they have to be lifted to the system net, which seems quite unnatural. By using nets-within-nets we can investigate the concurrency of the system and the agent in one model without loosing the abstraction needed.

Following [KMR03], we distinguish four different kinds of mobility, which are know as spontaneous, subjective, objective and consensual moves of the mobile agent (cf. Figure 2, where A is the agent net as a net token):

- Spontaneous Move: Neither the agent nor its environment initiate the transport. The movement can take place, but it is not enforced. No coupling of the environment and the agent is needed.
- Subjective Move: The agent itself initiates the movement, so agent and environment have to be coupled. This is described by the channel move, which has to be enabled in the agent. The movement takes places if the environment is able to execute it.
- Objective Move: The environment initiates the movement of the agent. The agent is forced to be transported. The initiative of the environment is modelled by the place travel ticket.
- Consensual move: Both the environment and the agent come to an agreement on the movement. This is modelled by a combination of the channel move, which has to be enabled in the agent, and the external condition modelled by the travel ticket.

The multi-agent architecture Mulan [KMR01] provides facilities for subjective moves as well as for objective moves.

4 An Example Model

As mentioned in the introduction structural analysis is useful for the system's as well as for the mobile agent's side.

Analysis of the Platform The case study is given in Fig. 3.[4] The parameter $n \in \mathbb{N}$ denotes the capacity of the public location. Here only the agent system (i.e. the system net N) is shown, since an agent (i.e. $\hat{N}$) cannot be restricted by platform in advance. The synchronisation relation is also omitted for the same reason.

There are three locations: pool, public, and private. The pool location is the initialisation area; the public area is open for any agent, while the private area has restricted access: It is allowed that many agents are simultaneously in the public location, but there has to be at most one agent in the private location. This prevents agents from being spied out like for a pooling booth. The transitions between the locations model movement, which are either objective or consensual (depending on the synchronisation relation).

[4] In fact it is just the object-net variant of the reader/writer problem.

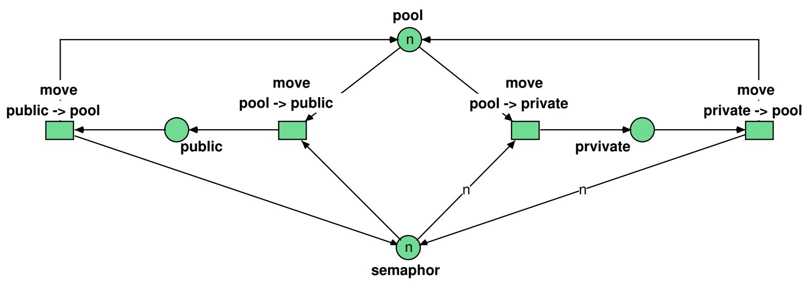


Fig. 3. The Multi-Agent System

In the following the system net N is analysed using invariants. The incidence matrix is given as:

$\mathit{\Delta} =$

	$pool \to pub$	$pub \to pool$	$pool \to prv$	$prv \to pool$
$pool$	-1	1	-1	1
$public$	1	-1		
$semaphor$	-1	1	$-n$	n
$private$			-1	1

Solving the equation $\boldsymbol{i} \cdot \mathit{\Delta} = \mathbf{0}$ we obtain $\boldsymbol{i}'_1 = (0, 1, 1, n)$ and $\boldsymbol{i}'_2 = (1, 0, -1, 0)$ as invariants of the system-net. Using Proposition 2 we have $\boldsymbol{i}_1 \cdot \mathit{\Pi}^1(\mathbf{M}) = \boldsymbol{i}_1 \cdot \mathit{\Pi}^1(\mathbf{M}_0)$ for all reachable markings $\mathbf{M}$:

$$\boldsymbol{i}_1 \cdot \mathit{\Pi}^1(\mathbf{M}) = M(public) + M(semaphor) + n \cdot M(public) = \boldsymbol{i}_1 \cdot \mathit{\Pi}^1(\mathbf{M}_0) = 0$$

and

$$\boldsymbol{i}_2 \cdot \mathit{\Pi}^1(\mathbf{M}) = M(pool) - M(semaphor) = \boldsymbol{i}_1 \cdot \mathit{\Pi}^1(\mathbf{M}_0) = n$$

Addition of these two linear equations results in:

$$M(public) + M(pool) + n \cdot M(private) = n$$

Therefore $M(private) > 0$ implies $M(private) = 1$ and $M(public) = M(pool) = 0$ which is the desired property.

Note, that property can be proven without having any knowledge about the structure of the agent (i.e. object net).

Analysis of an Agent In the following an example agent is analysed using invariants. The agent net N is given in Fig. 4. The two places flag_1 and flag_2 are used to toggle the agent's choice between the public and the private place.[5]

[5] Note, that due to this the agent activates the infinite firing sequence $w = ((move\ pool \to public)(public \to pool)(pool \to private)(private \to pool))^*$.

The incidence matrix is given as:

$\Delta =$

	pool $\longrightarrow$ *pub*	*pub* $\longrightarrow$ *pool*	*pool* $\longrightarrow$ *prv*	*prv* $\longrightarrow$ *pool*
ready public	-1	1		
public	1	-1		
ready private			-1	1
private			1	-1
flag$_1$	-1			1
flag$_2$		1	-1	

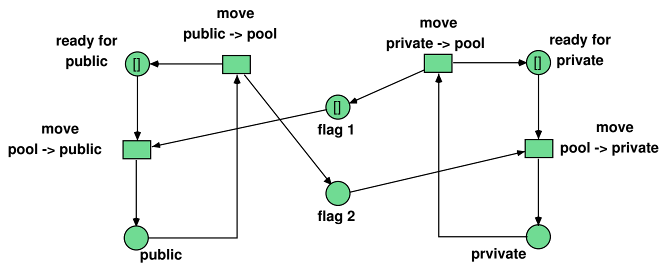


Fig. 4. The Agent

Solving the equation $\hat{\boldsymbol{i}} \cdot \Delta = \mathbf{0}$ we obtain $\hat{\boldsymbol{i}} = (1, 0, 1, 1, 1, 1)$ as the invariants of the agent-net. Using Proposition 2 we have $\hat{\boldsymbol{i}}_2 \cdot \Pi^2(\mathbf{M}) = \hat{\boldsymbol{i}}_1 \cdot \Pi^2(\mathbf{M}_0)$ for all reachable markings $\mathbf{M}$:

$$\begin{aligned} \hat{\boldsymbol{i}} \cdot \Pi^2(\mathbf{M}) &= \hat{M}(\mathit{public}) + \hat{M}(\mathit{private}) + \hat{M}(\mathit{flag}_1) + \hat{M}(\mathit{flag}_2) \\ &= \hat{\boldsymbol{i}} \cdot \Pi^2(\mathbf{M}_0) = 1 \end{aligned}$$

This implies:

$$\hat{M}(\mathit{public}) + \hat{M}(\mathit{private}) \leq 1$$

So, the agent proves that it does not attempts to enter the private and the public place at the same time.

5 Conclusion

In this presentation we have introduced the formalism of object nets and its invariant calculus, which has the compositionality property, i.e. invariants of the whole system are deducible from the components. The usefulness of structural analysis combined with compositionality of multi-agent systems is obvious in the context of mobility, since the system is open and only parts of the systems are known in advance. Using this approach we could establish a correctness proof for our example scenario without any knowledge about the structure of the mobile agents in the system.

It is our aim to model the *concepts* of mobile agents in terms of Petri nets, which is different from using Petri nets as a formalism to implement mobile agents, like in [XD00]. Especially, we tackle the general description of mobility as a concept on its own to have a direct focus on the main problems. So, mobile agents are just an example of the general approach to the concept of mobility. The formal model of nets-within-nets is used to define security properties of the system. The formal analysis is directly integrated into an existing multi-agent system architecture MULAN [KMR01].

References

[BOP00] Bauer, B., Odell, J., und Parunak, H. v. D.: Extending UML for Agents. In: *Proceeding of Agent-Oriented Information Systems Workshop.* S. 3 – 17. 2000.

[Bu99] Busi, N.: Mobile nets. *Formal Methods for Open Object-Based Distributed Systems.* S. 51–66. 1999.

[CGG99] Cardelli, L., Gordon, A. D., und Ghelli, G.: Mobility types for mobile ambients. In: *Proceedings of the Conference on Automata, Languages, and Programming (ICALP'99).* volume 1644 of *Lecture Notes in Computer Science.* S. 230–239. Springer-Verlag. 1999.

[Ch98] Chess, D. M.: Security issues in mobile code systems. In: Vigna [Vi98]. S. 1–17.

[Fa00] Farwer, B.: *Linear Logic Based Calculi for Object Petri Nets.* Logos Verlag. Berlin. 2000.

[KMR01] Köhler, M., Moldt, D., und Rölke, H.: Modeling the behaviour of Petri net agents. In: Colom, J. M. und Koutny, M. (Hrsg.), *International Conference on Application and Theory of Petri Nets.* volume 2075 of *Lecture Notes in Computer Science.* S. 224–241. Springer-Verlag. 2001.

[KMR03] Köhler, M., Moldt, D., und Rölke, H.: Modelling mobility and mobile agents using nets within nets. In: v. d. Aalst, W. und Best, E. (Hrsg.), *International Conference on Application and Theory of Petri Nets 2003.* volume 2679 of *Lecture Notes in Computer Science.* S. 121–140. Springer-Verlag. 2003.

[KR03] Köhler, M. und Rölke, H.: Concurrency for mobile object-net systems. *Fundamenta Informaticae.* 54(2-3). 2003.

[KR04] Köhler, M. und Rölke, H.: Properties of object Petri nets. In: Cortadella, J. und Reisig, W. (Hrsg.), *International Conference on Application and Theory of Petri Nets 2004.* Lecture Notes in Computer Science. S. 278–297. Springer-Verlag. 2004.

[Lo00] Lomazova, I. A.: Nested Petri nets – a formalism for specification of multi-agent distributed systems. *Fundamenta Informaticae.* 43(1-4):195–214. 2000.

[Sun97] Sun Microsystems. Secure computing with java: now and the future. 1997. http://java.sun.com/security/.

[MPW92] Milner, R., Parrow, J., und Walker, D.: A calculus of mobile processes, parts 1-2. *Information and computation.* 100(1):1–77. 1992.

[NL97] Necula, G. C. und Lee, P.: Safe, untrusted agents using proof-carrying code. In: *Special Issue on Mobile Agent Security.* volume 1419 of *Lecture Notes in Computer Science.* Springer-Verlag. 1997.

[ST98] Sander, T. und Tschudin, C. F.: Protecting Mobile Agents Against Malicious Hosts. In: Vigna [Vi98]. S. 44–60.

[Ts99] Tschudin, C. F.: Mobile agent security. In: Klusch, M. (Hrsg.), *Intelligent Information Agents – Agent based information discovery and management on the Internet.* chapter 18, S. 431–445. Springer-Verlag. 1999.

[Va98] Valk, R.: Petri nets as token objects: An introduction to elementary object nets. In: Desel, J. und Silva, M. (Hrsg.), *Application and Theory of Petri Nets.* volume 1420 of *Lecture Notes in Computer Science.* S. 1–25. 1998.

[VC98] Vitek, J. und Castagna, G.: Seal: A framework for secure mobile computations. In: *ICCL Workshop: Internet Programming Languages.* S. 47–77. 1998.

[Vi98] Vigna, G. (Hrsg.): *Mobile Agents and Security.* volume 1419 of *Lecture Notes in Computer Science.* Springer-Verlag. 1998.

[XD00] Xu, D. und Deng, Y.: Modeling mobile agent systems with high level Petri nets. In: *IEEE International Conference on Systems, Man, and Cybernetics'2000.* 2000.

CPN Models as Enhancement to a Traditional Software Specification for an Elevator Controller

Jens Bæk Jørgensen
Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
email: jbj@daimi.au.dk

Abstract

We demonstrate how Coloured Petri Nets (CPN) can be used to enhance a traditional software specifications document for an elevator controller. The traditional specification is given in a textbook. It includes a description of the desired functionality of the controller and a description of the subject domain (floors, buttons, cages, etc.). Based on the given specification, we build a CPN model, which is a coherent description that ties together different pieces of the given specification. The CPN model is used to argue for the correctness of the specification. Using simulation, we investigate a number of scenarios. For each scenario, we check that if a controller is implemented in compliance with the specification, it can ensure the desired effects in the subject domain (e.g., that each request can eventually be served).

Topics: Petri nets: embedding in traditional software engineering approaches; Modelling: methodologies, paradigms and principles.

1 Introduction

In this paper, we give an example on how *Coloured Petri Nets (CPN)* [13, 18] can be used to enhance a traditional software specification. We consider specification of an *elevator controller*.

The elevator controller must work in a ten floor building in which there are two elevator cages. The main responsibility of the controller is to control the movement of these cages. Movement is triggered by pushes on buttons. On each floor, there are buttons which can be pushed to call the elevator; there are both up and down buttons. Inside each cage, there are buttons, which can be pushed to request to be carried to a particular floor. In addition to controlling the movement of the cages, the controller is responsible for providing feedback to the elevator users. Inside each cage, there is a location indicator, which displays the current floor of the cage. On each floor, there is a direction indicator for each of the two cages showing the current direction of travel, if any. The controller is responsible for updating the location and direction indicators.

The elevator example have appeared often in the literature, e.g., in Jackson's work [9, 10, 11, 23]. Here, we use Wieringa's rendering. In [22], Wieringa gives a mission statement outlining the overall purpose of the controller, a dictionary fixing the meaning of terms in the subject domain, a set of entity-relationship diagrams of the subject domain, a set of statecharts [7] specifying the assumed behaviour of entities in the subject domain, and a set of statecharts specifying the desired behaviour of the entities to be controlled by the elevator controller. Moreover, the specification includes a number of alternative requirements-level architectures, represented as a kind of generalised data flow diagrams that outline the software structure of the controller together with the subject domain.

The contribution of this paper is to demonstrate how CPN can be used to enhance and argue for the correctness of the given specification. We create a CPN model that ties pieces of the given specification together and thus contributes to providing an overview. We explain how the CPN model serves as an alternative specification of a requirements-level architecture.

Many values gained by adding a CPN model are, of course, well-known. In particular, by creating and simulating a model, we gain insights and the model can be used as documentation. We will not comment these values much in this paper. Our focus is to make two main points. The first point is to explain the link between the given specification and the CPN model, e.g., how entities of entity-relationship diagrams are represented as tokens and how given statecharts are emulated in the CPN model. The second point is to describe how the CPN model is used to make what Wieringa calls *the system engineering argument*: to argue that under the given assumptions about the subject domain, with the current specification, certain desired properties will hold in the subject domain as controlled by the controller (e.g., that each request can eventually be served).

The remainder of this paper is structured as follows: Section 2 is excerpts of Wieringa's specification of the elevator controller. Section 3 describes the CPN model and discusses its relation to the given specification. Section 4 demonstrates how the CPN model is used to make the system engineering argument. Section 5 discusses more broadly a number of perspectives on using CPN in software engineering. Section 6 concludes the paper and includes a list of subjects for further discussion and investigations.

2 Given Specification

We now present excerpts of the specification of the elevator controller as presented i App. D of [22]. We describe (1) the desired functionality of the controller, (2) entities in the subject domain, and (3) the desired behaviour of the subject domain as controlled by the controller.

2.1 Desired Functionality

The *mission statement* outlines the overall purpose of the elevator controller: "to coordinate the movements of a number of elevator cages in order to provide optimal service to passengers waiting for service or travelling in the elevator".

The mission statement is supplemented with a *function refinement tree*, which specifies the functionality to be provided in a tree structure. The main functionality is: passenger support, maintenance support, installation support, and operation support. Wieringa restricts himself to consider the passenger support functions in the remainder of the case study; so do we. The passenger support functions are:

- collect passengers;
- deliver passengers;
- show floor;
- show direction.

These four functions are described in natural language in terms of their triggering events and the service delivered. We will discuss them in more detail in Sect. 4, where we make the system engineering argument.

The description of desired functionality is concluded with the partial *context diagram* reproduced in Fig. 1 (reproductions not completely identical to originals with respect to graphical layout).

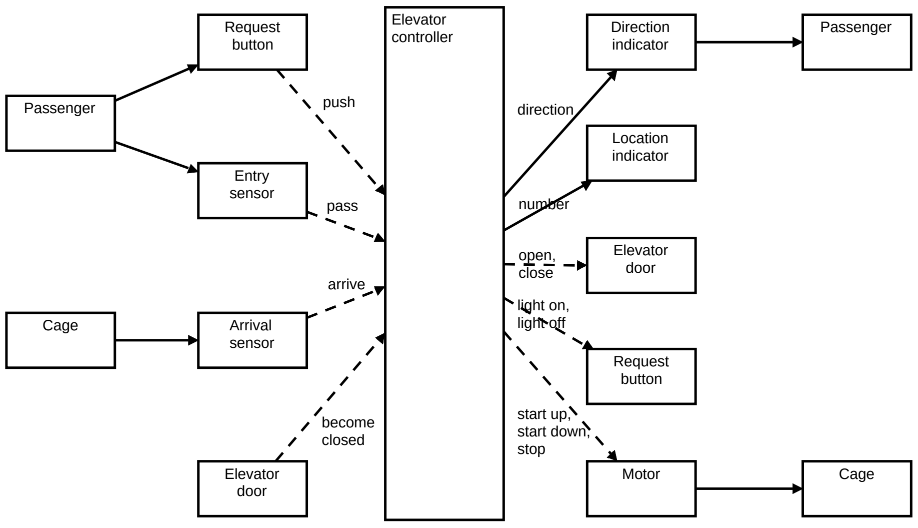


Figure 1: Partial context diagram.

The context diagram depicts the elevator controller and relevant entities in the subject domain. Also, the diagram indicates communication lines between different entities, e.g., that when a passenger pushes a request button, the elevator gets a `push` stimulus. The diagram indicates that the elevator controller may respond with, among other things, generating a `start up` response to the motor and a `light on` response to the request button.

2.2 Entities in Subject Domain

The subject domain specification includes a number of *entity-relationship diagrams (ERDs)* of parts of the subject domain. The diagram depicting entities related to an elevator cage is reproduced in Fig. 2.

The ERD shows entities that can exist in the subject domain. It also shows how many entities that can exist and how entities may be related. Thus, from Fig. 2, we can see that the elevator controller will work in an environment consisting of entry sensors, elevator doors, elevator cages, motors, location indicators, destination buttons, and floors. We can also see, e.g., that there are

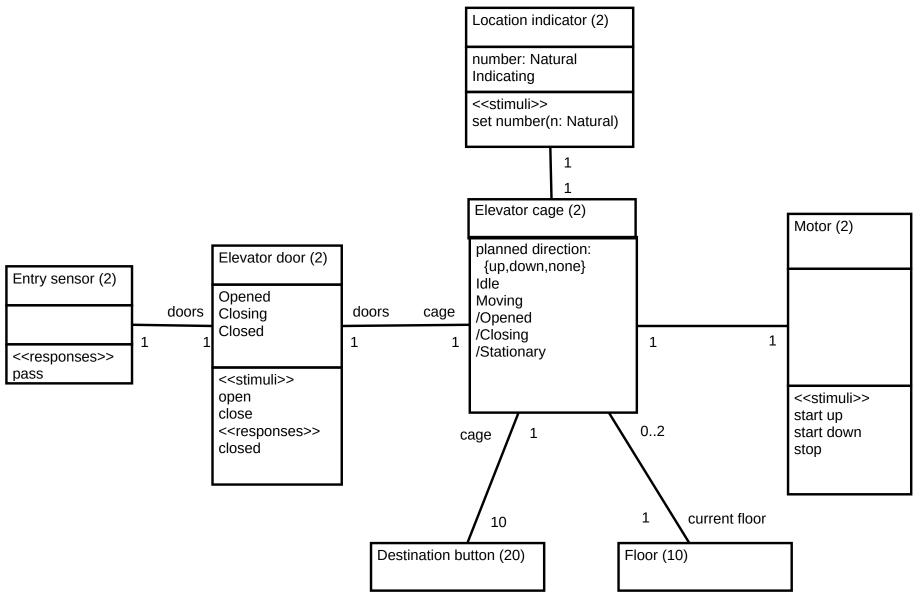


Figure 2: ERD for entities related to a cage.

exactly two elevator cages and exactly two elevator doors, and that each elevator door is related to exactly one elevator cage.

The entities in Fig. 2 are given; the elevator controller must be designed to work properly together with them, and with other given entities. Therefore, it is important that the designers of the elevator controller know exactly how the given entities work. The behaviour of direction indicators, location indicators, request buttons, elevator cages and motors, and elevator doors is specified in a set of *statecharts*. In the sense of the Robertsons' Volere requirements process [21], these statecharts give *constraints* for the elevator controller. In the sense of Jackson [10], the statecharts are *indicative descriptions*, which describe *assumed behaviour* of entities in the subject domain.

2.3 Desired Behaviour of Subject Domain

We now turn our attention to the *desired behaviour* of the subject domain to be brought about by the controller; this is an *optative description* [10]. A set of statecharts is used to specify the desired behaviour of the location indication process, of the allocation of requests to cages, and of the elevator cage movement. Figure 3 reproduces the statechart for desired elevator cage movement.

An example of the behaviour described in Fig. 3 is the following: The elevator cage is idle — the statechart is in the `Idle(c)` state. A request is made and the cage start to move — the statechart makes a transition from `Idle(c)` to `Moving(c)`, triggered by the event `allocate(b,c)` and assuming that the guard `[not Atfloor(b,c)]` is true. The cage arrives at a floor for which a request has been made and opens its doors. — the statechart makes a transition from `Moving(c)` to `Opened(c)`. Finally, the elevator cage's doors close and the elevator cage becomes idle again — the statechart makes two transition; first

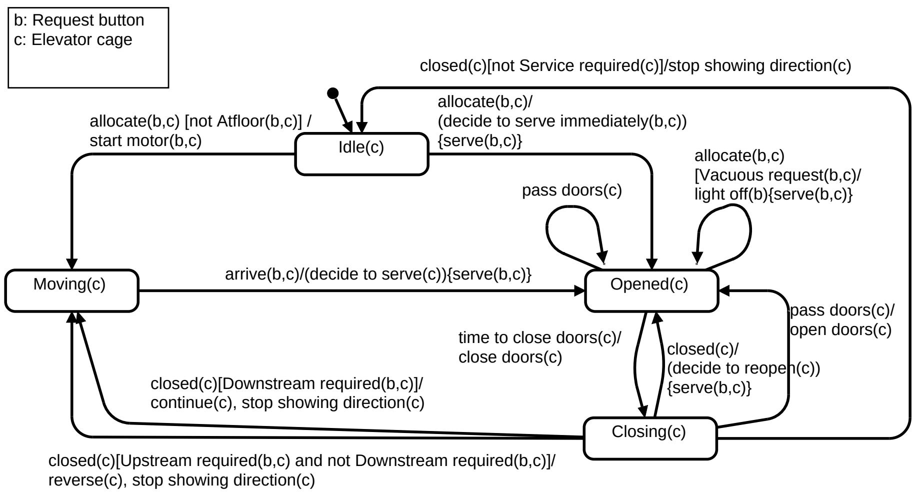

b: Request button
c: Elevator cage
closed(c)[not Service required(c)]/stop showing direction(c)
allocate(b,c)/
(decide to serve immediately(b,c))
{serve(b,c)}
allocate(b,c) [not Atfloor(b,c)] /
start motor(b,c)
Idle(c)
allocate(b,c)
[Vacuous request(b,c)/
light off(b){serve(b,c)}
pass doors(c)
Moving(c)
arrive(b,c)/(decide to serve(c)){serve(b,c)}
Opened(c)
pass doors(c)/
open doors(c)
time to close doors(c)/
close doors(c)
closed(c)/
(decide to reopen(c))
{serve(b,c)}
closed(c)[Downstream required(b,c)]/
continue(c), stop showing direction(c)
Closing(c)
closed(c)[Upstream required(b,c) and not Downstream required(b,c)]/
reverse(c), stop showing direction(c)

Figure 3: Statechart for desired behaviour of elevator cage movement.

from `Opened(c)` to `Closing(c)` and then from `Closing(c)` to `Idle(c)`.

The statechart of Fig. 3 is in [22] supplemented with tables specifying the details of the three complex transitions `decide to serve immediately`, `decide to serve`, and `decide to reopen`.

3 CPN Model

We now present the CPN model we have built based on the specification of the previous section; the model is built with *CPN Tools* [24].

We describe how the CPN model represents (1) entities in the subject domain, (2) the controller itself, and (3) the desired behaviour of the subject domain as controlled by the controller. We also explain how the CPN model serves as an alternative specification of a requirements-level architecture.

3.1 Representation of Entities in Subject Domain

Many entities in the subject domain are represented via colour set declarations. These declarations are derived directly from the ERD of Fig. 2 and other ERDs of [22], not shown in this paper. As examples, there are colour set declarations for the entities elevator cage, location indicator, destination button, and floor, which appear in Fig. 2.

The colour set used to represent the elevator cages consists of 4-tuples `(cageid,floor,requestlist,direction)`; `cageid` identifies the cage, `floor` is the number of the floor the cage currently is at (if the cage is stationary) or has last visited (if the cage is moving), `requestlist` is a list of the request that the cage is about to serve (represented as a list of floor numbers), and `direction` holds the current direction of the cage (up, down, or none).

3.2 Representation of Controller

The controller itself is represented as a set of functions. The control-movement functions are:

`setdirection`: called when the motor of an idle cage is about to start; determines whether the cage should move up or down, based on the current floor of the cage and its request list (it is assumed that the controller will maintain a request list for each cage recording the outstanding requests currently allocated to that particular cage).

`stophere`: called when the cage is in the final approach to a floor, triggered by a sensor; determines whether the cage should stop here. The cage stops if the floor number is in its request list; otherwise it continues.

`turnidle`: called when the cage's doors are closing; determines whether the cage should turn idle at the current floor. The cage should turn idle if it has no outstanding requests; otherwise, it should continue.

`servenow`: called when an idle cage with an empty request list receives its first request; determines whether that request can be served immediately. This is possible only if the request comes from the cage's current floor. In this case, the cage can just open its doors; it is not necessary to start the motor.

`resetdirection`: called when the doors of a cage with outstanding requests have closed; determines whether the cage should move up or down, based on the current direction of the cage and its request list. If there are any requests in the list that can be served in the current direction, the cage continues. Otherwise, it shifts to moving in the opposite direction.

`addrequest`: called when a new request is made; adds a request to the request list of a cage.

`removerequest`: called when a cage opens its doors at a certain floor. The requests corresponding to that floor are removed from the cage's request list.

The elevator controller should also supply two information-display functions `updatelocationindicators` and `updatedirectionindicators`. However, we have only included the first of these in the CPN model. The second could easily be added, but we have chosen not to include update of direction indicators in the current version of the model in order not to clutter up the graphics.

3.3 Desired Behaviour of Subject Domain

The desired behaviour of the subject domain as controlled by the controller is described via net structure and comprises the following three modules:

- `Basic Cage Movement`;
- `Requests and Allocation`;
- `Up/down and Indicators`.

Basic Cage Movement

The `Basic Cage Movement` module is shown in Fig. 4

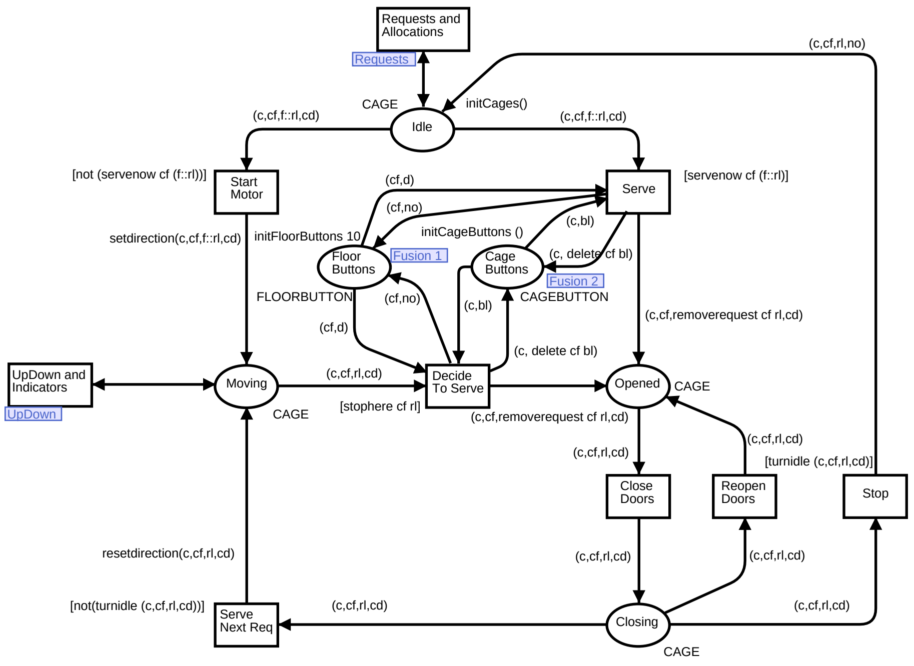


Figure 4: Basic Cage Movement module of CPN model.

The major part of Fig. 4 is the result of a straightforward manual translation of the statechart of Fig. 3. The states of the statechart have been translated into places in the CPN model. As example, the states `Idle(c)` and `Moving(c)` of the statechart are translated into the places `Idle` and `Moving` in the CPN model, respectively. The events, actions, and guards have been translated into transitions surrounded with arcs with appropriate inscriptions. As example, the action `start motor(b,c)` appearing in the statechart on the arc from `Idle(c)` to `Moving(c)` is translated into the transition `Start Motor` of the CPN model.

In the statechart, the action `start motor(b,c)` has the triggering event `allocate(b,c)`, which corresponds to the request `b` being allocated to the cage `c` in the subject domain. Moreover, `start motor(b,c)` has the guard `[not Atfloor(b,c)]`, which corresponds to the elevator cage `c` not currently being at floor `b` in the subject domain. The CPN model models allocation of request `b` to cage `c` in the subject domain by occurrence of a transition (on the `Requests and Allocation` module), which causes a token to appear on the `Idle` place, which matches the pattern `(c,cf,f::rl,cd)` appearing on the arc from `Idle` to the `Start Motor` transition. The guard of the transition `Start Motor` in the CPN model being true models that the cage `c` is not currently at floor `b` in the subject domain.

In this way, the arc from `Idle(c)` to `Moving(c)` in the statechart corresponds to the same subject domain phenomena as the transition `Start Motor` of the CPN model.

The two substitution transitions `Requests and Allocations` and `UpDown and Indicators` in Fig. 4 have no counterparts in the statechart of Fig. 3. These transitions tie the `Basic Cage Movement` module of the CPN model together with other modules, and this is not done in the statechart. The `Idle` socket place is bound to a port place with the same name on the `Requests and Allocation` module and the `Moving` socket place is bound to a port place with the same name on the `Up/down and Indicators` module.

Also, the two places `Floor Buttons` and `Cage Buttons`, used to model the floor buttons and the cage buttons, have no counterparts in the shown statechart. We will discuss them further below.

Requests and Allocation

The `Requests and Allocation` module is shown in Fig. 5.

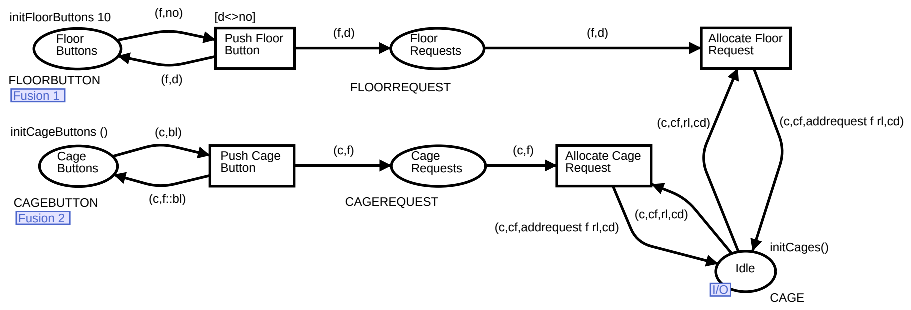


Figure 5: Request and Allocation module of CPN model.

Each of the two places `Floor Buttons` and `Cage Buttons` are fused with the place with the same name on the `Basic Cage Movement` module.

Occurrence of the `Push Floor Button` transition models that a passenger pushes a floor button, either in the direction up or down. In the subject domain, this will cause the floor button to light up, modelled in the CPN model by an update of a floor button token on the `Floor Buttons` place (that the light of the floor button is eventually turned off again is modelled by occurrence of one of the transitions `Serve` or `Decide To Serve` on the `Basic Cage Movement` module (Fig. 4)).

The occurrence of `Push Floor Button` also causes a `FLOORREQUEST` token to be added to the `Floor Requests` place. Subsequently, this may cause the transition `Allocate Floor Request` to become enabled. It is the `Allocate Floor Request` transition, which models assignment of a given floor request to one of the two elevator cages. In the current version of the model, the scheduling policy is very simple: It models that a random idle elevator is chosen. We discuss more advanced scheduling policies in Sect. 5.4.

The `Push Cage Button` transition models that a passenger pushes a cage button inside an elevator cage. Similarly to `Push Floor Button`, this causes an update of a token on the `Cage Buttons` place modelling that the button lights up. It also causes a `CAGEREQUEST` token to appear on the `Cage Requests` place. Occurrence of the transition `Allocate Cage Request` models that the request is added to the request list of the cage in which the button is pushed.

Up/down and Indicators

The `Up/down and Indicators` module is shown in Fig. 6.

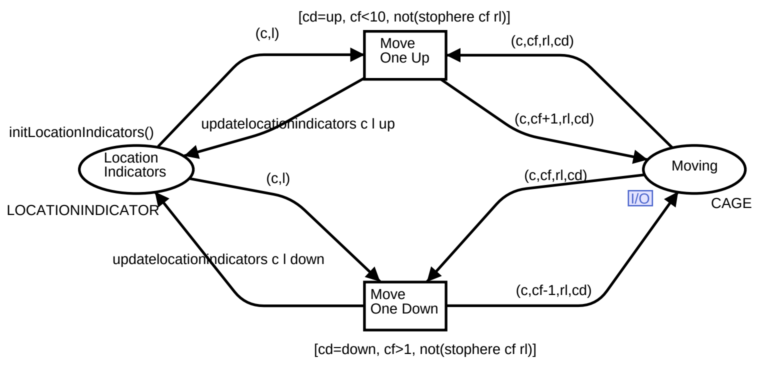


Figure 6: Up/down and Indicators module of CPN model.

The `Moving` place is conceptually glued with the place with the same name in the `Basic Cage Movement` module.

The `Move One Up` transition models movement of an elevator cage one floor up. When it occurs, the `floor` entry of the `CAGE` token on `Moving` is incremented. The guard of `Move One Down` consists of three Boolean expressions and should be read as a conjunction. The expressions `cd=up` and `cf<10` models properties that are inherent to the subject domain: that the cage actually is moving upwards and that it cannot move up if it is at the topmost floor. The expression `not(stophere (c,cf,rl,cd))` models a condition that is enforced by the elevator controller: that the cage only moves if it should not stop — it should stop if there is a request. The `Move One Down` transition works similarly to `Move One Up`.

Tokens on the `Location Indicators` place model the location indicators for the two elevator cages. The state of `Location Indicator` is updated each time either `Move One Up` or `Move One Down` occurs. The call of the function `updatelocationindicators` emulates the controller's responsibility of ensuring that each location indicator correctly displays the actual current floor of the elevator cage. In terms of the CPN model, this means that the `floor` entry of each of the tokens on `Location Indicators` must be equal to the `floor` entry of the corresponding `CAGE` token — which is on one of the places `Idle`, `Moving`, `Opened`, or `Closed` (this property could be formally described using a place invariant).

3.4 CPN Model as Requirements-level Architecture

In Wieringa's terms, a *requirements-level architecture* is a kind of generalised data flow diagrams that outline:

1. the processes and data stores that make up a system;

2. entities in the subject domain;

3. possible internal communications in the system;
4. possible communications between the system and entities in the environment.

A requirements-level architecture is determined by, indeed, the requirements to the system to be developed and by the environment of use. It is a description, which is independent of any specific implementation platform.

Wieringa presents two different requirements-level architectures for the elevator controller. Both architectures have resemblance with the context diagram reproduced in Fig. 1 in the sense that the architecture diagrams show entities of the subject domain plus the controller itself. The difference is that in Fig. 1, the controller is shown as a black box, whereas the architecture diagrams also show the internal structure of the controller. In the Von Neumann architectural style (Fig. D.20 of [22]), data storage and data processing are kept separate; data stores never process data themselves and data transformations have no memory to store data. In the subject-oriented decomposition, object-oriented architectural style (Fig. D.24 of [22]), the internal structure of the controller is made up of objects that encapsulate behaviour and data transformations that correspond to aspects of the subject domain.

The CPN model may be seen as an alternative requirements-level architecture in the following way:

1. The processes that make up the elevator controller are represented by the functions described in Sect. 3.2. The data stores of the system appear in tokens such as the `requestlist` component of `CAGE` tokens.
2. Entities in the subject domain of the controller are represented via colour set declarations and tokens as described in Sect. 3.1.
3. Possible internal communications in the controller are described via transitions. As examples, the two transitions `Allocate Floor Request` and `Allocate Cage Request` on the `Requests and Allocation` module shown in Fig. 5 model the allocation of requests to cages. Occurrence of these transitions correspond to internal communications.
4. Possible communications between the controller and entities in the environment are also described via transitions. As examples, the two transitions `Push Floor Button` and `Push Cage Button` on the `Requests and Allocation` module correspond to communications between buttons (external entities) and the controller.

4 The System Engineering Argument

As discussed in the introduction, the system engineering argument consists in arguing that under the given assumptions about the subject domain, with the current specification, certain desired properties will be true for the subject domain as controlled by the controller.

We now describe (1) that the CPN model gives prerequisites for making the argument, (2) the desired properties we are considering, and (3) how the argument is made using simulation.

4.1 Prerequisites for the Argument

The prerequisites for the argument are two basic properties of the CPN model:

- The CPN model is a coherent description (as we will argue below).
- The CPN model is executable.

The CPN model is a coherent description in the sense that it ties together different pieces of Wieringa's specification. In the first place, it ties together descriptions of entities in the subject domain with descriptions of behaviour. As an example, large parts of the ERD of Fig. 2 and the statechart of Fig. 3 are brought together in the CPN model. The statechart is emulated in the `Basic Cage Movement` module of the CPN model, and entities of the ERD appear as tokens in the CPN model.

Secondly, the CPN model ties together a number of Wieringa's statecharts behavioural descriptions. As an example, the statechart describing assumed behaviour of request buttons (Fig. D.8 of [22], not shown in this paper) is reflected in the `Requests and Allocations` module of the CPN model, shown in Fig. 5. As we have seen previously, the desired behaviour of elevator cage movement is represented both in the statechart of Fig. 3 and in the module of the CPN model shown in Fig. 4. In this way, the CPN model ties together the given behavioural descriptions for request buttons and for movement of elevator cages. Thus, the CPN model describes causalities that the elevator controller must deal with — the pushing of a request button generates an event, which may trigger that an elevator cage starts to move.

4.2 Desired Properties

The desired properties that we must consider to make the system engineering argument for the elevator controller are provision of the passenger support functions mentioned in Sect. 2.1: collect passengers, deliver passenger, show floor, and show direction.

Let us consider Wieringa's description of the first of these properties:

Name: Collect passengers

Triggering event: A passenger pushes a floor button at floor F.

Delivered service: The controller ensures that an elevator cage stops at floor F and allows passengers to enter. Average round-trip time when a cage is used at 80% of the guaranteed load should be no more than 60 seconds.

Thus, to make the system engineering argument, we must argue that each time a passenger pushes a floor button at floor F, the controller ensures that an elevator cage eventually stops at floor F and allows passengers to enter (estimating round-trip times is outside the scope of this paper).

4.3 Making the Argument via Simulation

We use simulation to investigate different scenarios and check that in each one, when the model emulates that a passenger pushes floor button F in the subject domain, eventually the model can be in a state, which corresponds to an elevator cage being present at floor F with its doors open. As an example, consider a situation in which both elevator cages are idle at the first floor. There are no outstanding requests. Let's say that the floor button in pushed at the 4th floor; the passenger wants to travel up.

In the model, the corresponding event is that the transition `Push Floor Button` (see Fig. 5) occurs with its `FLOOR` variable `f` bound to the value `4` and its `DIRECTION` variable `d` bound to the value `up`. The assumed subject domain situation implies that two `CAGE` tokens are present at the `Idle` place (both have empty request lists). Therefore, the transition `Allocate Floor Request` becomes enabled. When it occurs, we can see in the model that a request from the 4th floor is properly added to the request list of one of `CAGE` tokens on the `Idle` place, say cage `cg(1)`, by the function `addrequest`.

In the current marking, the transition `Start Motor` (Fig. 4) is enabled; the guard `[not (servenow cf (f::rl))]` is true because `cf=1` (the cage's current floor) and `f=4` (the number of the floor on which the considered request was made), and `servenow` just checks if `cf` is equal to `f`. When `Start Motor` occurs, the `CAGE` token for cage `cg(1)` is removed from the `Idle` place and added to the `Moving` place. We can check that the direction of movement is correctly set to `up`, by the expression `setdirection(c,cf,f::rl,cd)` because `cf=1` and `f=4` and setdirection results in `cd=up` when `cf` is less than `f`.

The transition `Move One Up` (Fig. 6) is now enabled; the guard `[stophere cf rl]` evaluates to false because `cf=1` and `rl=[4]` and `stophere` just checks if `cf` is member of `rl`. The occurrence of `Move One Up` causes the `floor` entry in the `CAGE` token on the `Moving` place to change from `1` to `2`. In general, each time `Move One Up` occurs, the `floor` entry is incremented. Thus, when `Move One Up` has occurred two additional times, the `floor` entry is `4`. Now, the guard `[stophere cf rl]` evaluates to true because `4` is member of `[4]`. Therefore, the transition `Decide To Serve` (Fig. 4) becomes enabled. When it occurs, the `cg(1) CAGE` token is removed from the `Moving` place and added to the `Opened` place. This corresponds to the subject domain event that elevator cage number one arrives at the 4th floor and opens its doors; passengers are allowed to enter, and, thus, in this case, the desired property can be ensured.

We have simulated a number of scenarios, also much more complex scenarios than the one we have described in detail here. In each case, we have observed that any time a request made in the model, we can bring the model in a state, which corresponds to the request eventually being served. In this way, we have used simulation to make the system engineering argument for a number of scenarios and thus to increase our confidence of the specification.

We can make similar arguments for the other desired properties (except the last one regarding direction indicators, which we have not included in the current version of the model). In all cases, we can observe, via simulation, that the desired behaviour of the subject domain can be achieved for the considered scenarios with the current specification of the controller.

5 Perspectives on CPN in Software Engineering

In this section, we go from considering the elevator controller example to a discussion of more general perspectives on the use of CPN in software engineering, including use together with more traditional software development methods.

5.1 Problem Frames

Wieringa's emphasis on and rendering of the system engineering argument is highly inspired by Jackson's decades-long work on the relationship between the real world and the computer systems (machines) we develop.

The basic perspective in Jackson's newest book on *problem frames* [10] is that software development is an undertaking in which problems in the real world are solved with the help of machines. Various problems in software engineering can sometimes be fit into a particular frame, which can be used to analyse and structure the description of a particular problem. Problem frames are a problem-side counterpart to the more well-known design patterns of object-oriented software development [6] on the solution-side.

The control-movement part of the elevator controller problem we have considered in this paper fits the *commanded behaviour problem frame*, and the system engineering argument we have made is an address of *the frame concern.* In general, this consists in the first place of making appropriate descriptions of the problem domain (subject domain), the requirements to be satisfied, and the specification of a machine. Secondly, based on these descriptions, a convincing argument must be given that the machine and the given domain properties together entail that the requirements are fulfilled.

This is what we have done for the elevator controller in this paper. We described the problem domain in Sects. 2.2 and 2.3. We looked in at the requirements to be satisfied, first in Sect. 2.1 and then in more detail in Sect. 4.2. We specified the machine in Sect. 3.2. In Sect. 4, we argued that if an elevator controller is implemented, which is compliant with the specification, and if the given entities of the problem domain (buttons, doors, motors, etc.) work as assumed, then the requests we have considered can eventually be served.

The system engineering arguments for the show floor and show direction requirements (see Sects. 2.1 and Sect. 4.2) fit the *information display problem frame.*

We have previously used CPN models to make system engineering arguments in [15]. Here, we addressed the frame concern for a commanded behaviour problem in a complex biddable domain [10] involving people: support of the work process medicine administration as carried out by nurses working at a hospital. Thus, in summary, CPN is useful to address the frame concern in various problem frames.

5.2 Jackson's Basic Tenets

In continuation of the discussion of Jackson's problem frames above, we now consider three basic tenets for making descriptions in software development, stated by Jackson in [11]. These are: *1. Distinguish the machine from the problem domain; 2. Don't restrict description to the machine; 3. State explicitly what is described.*

With the descriptions given in this paper, Wieringa and we adhere to these tenets. In particular, the elevator controller is an embedded system and as such only visible to its users via the effect it has in the subject domain. Therefore, we have emphasised giving descriptions of the subject domain, in accordance with tenet 2. However, we have realised tenet 2 fundamentally different from the recommendation of [11], which is to make separate descriptions of on one hand the system to be built and on the other hand the environment in which the system will be used. Not making the separation is seen by Jackson as a major cause of confusion. The risk is that desired properties of the system under design are mixed up with inherent and given properties of the environment.

While we agree that it is important to distinguish between indicative and optative descriptions, we disagree that it is necessary to keep the descriptions of the environment and the description of the machine separate. On the contrary, we believe that it is possible and indeed quite useful to gather descriptions of the environment and the machine in one single integrated description. In fact, this is what we have done with the CPN model.

The CPN model is an optative description; it describes a future, desired behaviour in the subject domain. It is the responsibility of the designers of the controller to specify controller functions such as `setdirection` in such a way that the desired effect in the subject domain is achieved.

5.3 Executable Software Specifications

The work in this paper is inspired by our work on *Executable Use Cases* [14, 16]. Executable Use Cases are an approach to requirements engineering, in which new work processes and their proposed computer support are represented via three tiers. Tier 1 consists of traditional UML-style use cases [3, 12], tier 2 is a formal model (e.g., a CPN model) built from tier 1, and tier 3 is a graphical animation based on tier 2. An executable use case enables stakeholders to interactively investigate system requirements in the context of the envisioned work processes.

This can be generalised to *executable software specifications*. Instead of merely having a natural-language use case at tier 1, in executable software specifications, we can have any given specification for a software system. The given specification may encompass, e.g., ERDs and statecharts describing assumed and desired behaviour as we saw for the elevator controller. The role of tier 2, the CPN model, is to be a a coherent and executable description, which formalise and tie together pieces from tier 1.

If tier 3, a graphical animation, is added, we get support for communication with clients with a non-technical background, if desired (this is not necessarily relevant for the elevator controller). In general, tier 3 makes it possible to view the executable software specification as a *context-descriptive prototype* [2].

Executable software specifications might candidate to be used in development projects tailored from the widely used *Rational Unified Process (RUP)* [19]. RUP emphasises the use of UML models as key artifacts in software development. However, if a deviation from UML is acceptable by the project stakeholders, creating and executing CPN models fit well in the elaboration phase or the inception phase of RUP.

5.4 Statecharts and CPN

Relationship between traditional software specifications in the form of various UML diagrams and CPN are discussed in [5] and [17]. In both these papers, CPN models are seen as alternatives to UML state machines or statecharts.

More generally, a number of authors have compared and proposed combined use of statecharts and various kinds of Petri nets, see, e.g., [4, 20].

For the elevator controller, if we want to use the specification as a basis for experimentation or prototyping, there seems to be an inherent need to go beyond behavioural descriptions in the form of statecharts. The reason is that a key problem that must be a solved by the elevator controller is the scheduling problem: Given a request, a decision must be made about which of the two elevator cages that should handle the request.

Wieringa's specification is not in itself a proper basis for experiments. The statechart of Fig. 3 models the life cycle of one single elevator cage. In order to have a basis for specifying and designing scheduling, we need to consider the two cages at the same time. This is done in the CPN model. The CPN model can conveniently be used to experiment with and investigate the consequences of different proposals for scheduling policies. In the current version of the model, as we saw in Sect. 3.3, any floor request is assigned to a randomly chosen idle cage. By adding a guard to the transition `Allocate Floor Request` (see Fig. 5), we could instantly change the policy so that, e.g., if a request is made on floor f, and cage c is idle at floor f, then f is assigned to c. Other straightforward scheduling policies could be to ensure that a floor request was assigned to the nearest cage or to the cage with the shortest request list. We could investigate consequences of considered changes via simulation.

Another point in the comparison of statecharts and CPN is that CPN's extensive state concept is an advantage in the specification of the elevator controller. As an example, it is in CPN very convenient to maintain lists of requests. Doing so with statecharts is awkward and does not scale well (a similar observation for another example is reported in [17]).

6 Conclusions and Discussion

It might be possible to base the system engineering argument for the elevator controller on Wieringa's specification alone — although Wieringa does not make the argument himself in [22]. Reconsider the two prerequisites for making the system engineering argument in Sect. 4.1: that the CPN model is executable and coherent. We can observe in the first place that with proper tool support, e.g., the Statemate tool [8], it is straightforward to make the statecharts of Wieringa's specification executable. In the second place to make a convincing system engineering argument seems to assume that the given statecharts can be seen as a set of interacting and communicating statecharts (Wieringa does not guide us to do so in [22]). However, if this assumption is true, some description, e.g., in prose, can be made like: if this action happens in this statechart, then that action happens in that statechart, etc. (similarly to what we did for the CPN model in Sect. 4.3)

In comparison, we believe that a system engineering argument based on a CPN model is relatively simple and reliable. If we in addition keep the general advantages of CPN in mind, there are certainly benefits obtained by adding a CPN model to the given specification.

On the other hand, it is still an open question whether these benefits outweighs the cost of creating and simulating the model. The alternative we must compare with is to start implementing the controller based on Wieringa's specification alone. To investigate which alternative is best would require a substantial and difficult effort. Moreover, it would be difficult to estimate to which extent the findings generalise to other similar development projects. However, if we want to promote the CPN-based approach, we should be ready to address a number of concerns, some of which we will discuss below.

6.1 Gap Between Model and Implementation

A very general concern often raised when model-based development is discussed is the inherent gap between an abstract specification or model and a specific implementation. This gap is the main reason to the current momentum of the *Extreme Programming (XP)* [1] approach to software development, with its principle of not, or only very scarcely, using models.

The CPN model merely is an abstract specification of the elevator controller in terms of a set of Standard ML functions (and the effect that the specification has in the subject domain). There is a gap between the specification and an implementation of the controller, both because (1) the CPN model does not cover all relevant issues that the implementation must eventually deal with, e.g., the logics in handling of failures of external entities like motors, doors, and sensors; but also because (2) the CPN model does not deal with the implementation platform for the controller.

Issue (1) above can be dealt with by more work on the CPN model. Issue (2), however, is difficult. How to take the step from a CPN model to a real implementation is a possible subject for future research.

6.2 Size and Quality of Given Specification

The size of the given specification is a factor to take into account if we consider adding a CPN model. Although it has been demonstrated convincingly that large CPN models can be built and executed (see, e.g., Vol. 3 of Jensen [13]), scalability of the applicability of CPN to make the system engineering argument is a genuine concern.

The elevator controller example is relatively simple, although sufficiently complex, we believe, to justify making an addition to the given specification allowing us to tie pieces together, to get an overview, and to provide support for making the system engineering argument. But what if the given specification consists of hundreds or thousands of pages? Does it then at all make sense to try to make a clean, academic system engineering argument? Of course, at least, it requires that the argument somehow can be made in a modular fashion.

The quality of the given specification is another factor: The given specification for the elevator controller is of a quality worth to make it part of a textbook and as such very well worked out, mature, and consistent. This made it straightforward to build the CPN model. However, this is not the kind of specifications normally found in real development projects. On the contrary, real-world specifications often have significant lacks, inconsistencies, flaws, etc. In such cases though, making a CPN model is a possible means to improve the quality of the specification (but again, whether it is worthwhile to do depends on a cost-benefit analysis in each specific project in which the option is considered).

6.3 Improvement of System Engineering Argument

A subject for future work is to use the elevator controller CPN model to make not only a system engineering argument, but a possibly more convincing *system engineering proof.*

State space analysis [13] may enable a more solid address of the frame concerns than we did in Section 4. Here, we merely argued that for each considered request, it is possible to bring the CPN model in a state corresponding to the request being served; we did not argue that this does always happen. Moreover, we may be able to analyse more advanced behavioural properties than is possible with simulation, such as fairness properties, e.g., the possibility of requests being starved. However, if we want to pursue state space analysis, we must probably do it for a simplified version of the model in which only a few button pushes are allowed and the building has only a few floors — and even perhaps there is only one elevator cage. State space analysis is hampered by the state explosion problem and does not scale well to large problems.

Whether a formal verification of a simplified version of the model or a simulation-based argumentation of the many button pushes, ten-floor, two-cage CPN model, which describes the real situation to be dealt with, is the best way to carry out the system engineering argument is a debatable point.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 1999.

[2] C. Bossen and J.B. Jørgensen. Context-descriptive Prototypes and Their Application to Medicine Administration. In *Proc. of DIS 2004*, pages 297–306, Cambridge, Massachusetts, 2004. ACM.

[3] A. Cockburn. *Writing Effective Use Cases.* Addison-Wesley, 2000.

[4] Z. Dong and Xudong He. Integrating UML Statechart and Collaboration Diagrams Using Hierarchical Predicate Transition Nets. *Lecture Notes in Informatics, vol. P-7*, pages 99–112, 2001.

[5] M. Elkoutbi and R.K. Keller. User Interface Prototyping Based on UML Scenarios and High-Level Petri Nets. In *Proc. of the 21st Petri Nets Conf.*, volume 1825 of *LNCS*, pages 166–186, Aarhus, Denmark, 2000. Springer-Verlag.

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

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[8] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The STATEMATE Approach.* McGraw-Hill, 1998.

[9] M. Jackson. *Software Requirements & Specifications — a lexicon of practice, principles and prejudices.* Addison-Wesley, 1995.

[10] M. Jackson. *Problem Frames — Analyzing and structuring software development problems.* Addison-Wesley, 2001.

[11] M. Jackson. Some Basic Tenets of Description. *Software and Systems Modeling*, 1(1):5–9, 2002. Springer.

[12] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.

[13] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1-3.* Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 1992-97.

[14] J.B. Jørgensen and C. Bossen. Requirements Engineering for a Pervasive Health Care System. In *Proc. of 11th International Requirements Engineering Conf.*, pages 55–64, Monterey Bay, California, 2003. IEEE.

[15] J.B. Jørgensen and C. Bossen. Executable Use Cases as Links Between Application Domain Requirements and Machine Specifications. In *Proc. of 3rd International Workshop on Scenarios and State Machines (at ICSE 2004)*, pages 8–13, Edinburgh, Scotland, 2004. IEE.

[16] J.B. Jørgensen and C. Bossen. Executable Use Cases: Requirements for a Pervasive Health Care System. *IEEE Software*, 21(2):34–41, 2004.

[17] J.B. Jørgensen and S. Christensen. Executable Design Models for a Pervasive Healthcare Middleware System. In *Proc. of the 5th UML Conf.*, volume 2460 of *LNCS*, pages 140–149, Dresden, Germany, 2002. Springer-Verlag.

[18] L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.

[19] P. Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, 1999.

[20] R.G. Pettit and H. Gomaa. Modeling State-Dependent Objects using Coloured Petri Nets. In *Proc. of MOCA'01*, pages 105–120, Aarhus, Denmark, 2001.

[21] S. Robertson and J. Robertson. *Mastering the Requirements Process.* Addison Wesley, 1999.

[22] R.J. Wieringa. *Design Methods for Reactive Systems: Yourdon, Statemate, and the UML.* Morgan Kaufmann, 2003.

[23] P. Zave and M. Jackson. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, 1997.

[24] CPN Tools. www.daimi.au.dk/CPNtools.

Modelling and Verification of Compatibility of Component Composition

Donald C. Craig and Wlodek M. Zuberek
Department of Computer Science
Memorial University of Newfoundland
St. John's, Canada A1B 3X5

Abstract

Two components are compatible if any sequence of operations requested by one of these components can be provided by the other component. If the set of all requested sequences is denoted by $\mathcal{L}_R$ and the set of all provided sequences of operations by $\mathcal{L}_P$, then the two components are compatible if $\mathcal{L}_R \subseteq \mathcal{L}_P$. This paper uses Petri nets to model the interface behaviours of interacting components (*i.e.* the languages $\mathcal{L}_R$ and $\mathcal{L}_P$) and formally defines the composition of components. Compatibility of components is verified by checking if the composed models contain deadlocks. Simple examples illustrate the proposed approach.

Keywords : component compatibility, component interfaces, software architecture, component-based software, Petri nets, deadlock detection

1 Introduction

The difficulties involved in the development of large-scale software architectures are well documented and, over the years, numerous strategies have been developed to help mitigate these difficulties [1]. Object-oriented programming [2] and numerous architectural description languages [3] have been introduced in order to make the development of software systems more tractable. During recent years, component based software engineering (CBSE) has been emerging as viable means of software construction whereby pre-manufactured software substructures with well-defined interfaces are designed and implemented and subsequently incorporated into larger software systems [4]. While this approach has met with some degree of success, there remains the problem of determining compatibility between components.

Classical techniques of determining compatibility have typically focused on compile-time metrics such as consistency between the numbers and types of method arguments and on appropriate use of a method return type. While such static checks are clearly important, they are insufficient in establishing the dynamic or behavioural compatibility between two software components. For example, it is possible for a server component to provide methods that exactly match the method requirements of a client component. However, if the service component imposes a rigid ordering upon the sequence of these method calls that are not adhered to by the client, it is still possible for the two components to exhibit conflicting behaviours. Such conflicts result in component incompatibility.

This paper provides the foundation for a formal model of component interaction by representing component interfaces using Petri nets [5, 6]. Interface compatibility is established by determining those interfaces that, when connected, are free of deadlock. The "requires" and "provides" relationships will be discussed in the context of the intersection of formal languages generated by the corresponding Petri nets in a component's deployment environment. By treating component behaviour as a language, compatibility between components is tested and verified.

A model of component interfaces that employs Petri nets and the notion of interface languages are introduced in Section 2. Section 3 describes the composition of component interfaces and provides a formal framework for establishing compatibility between two components using the Petri net model and deadlock detection. In Section 4, some examples that demonstrate the model are provided. Finally, Section 5 provides some applications of the model and discusses future work.

2 Petri Net Component Models

Several attempts have been made to define a component: many of these attempts have been summarized in [7]. Informally, a component can be thought of as a cohesive logical unit of abstraction with a well-defined interface that provides services to its environment. In order to behave correctly, the component would also likely require the services of other components in its environment. Some attempts have been made to formally define a component and its behaviour. Such definitions have even made extensive use of Petri nets [8].

For the purposes of this model, the low-level, internal behaviour of the component will be disregarded as it is not important in the formalism discussed below. The focus of attention is on the behaviour at the scope of the components' interfaces and not the internal dynamics of the components themselves. While it is certainly true that there may be an inseparable relationship between a component's internal behaviour and the dynamics manifested at the component's interface, this model will concentrate only upon the interface itself. The relevant behavioural properties that are necessary to ensure compatibility between components manifest themselves at the components' interface, thereby rendering communications that are strictly internal to the component irrelevant for the purposes of this discussion. Therefore, this proposal is not so much a model of a component as it is a model of a component's interface.

2.1 Interface Models

A component's interface is defined in terms of a labelled Petri net:

$$\mathcal{M}_i = (P_i, T_i, A_i, S_i, \ell_i, m_i)$$

In this definition, P_i and T_i are disjoint sets of places and transitions, respectively, $A_i \subseteq P_i \times T_i \cup T_i \times P_i$ is a set of directed arcs, S_i is an alphabet representing a set of services which are associated with labelled transitions, $\ell_i : T_i \to S_i \cup \{\varepsilon\}$ is a labelling function (ε is the empty label), finally, $m_i : P_i \to \{ 0, 1, \dots \}$ is the initial marking function. The Petri net model representing an individual interface must be deadlock-free. Although it is not necessary for the presented approach, it is assumed that the interfaces are represented by cyclic nets. A simple example of an appropriately marked and labelled interface is presented in Figure 1.

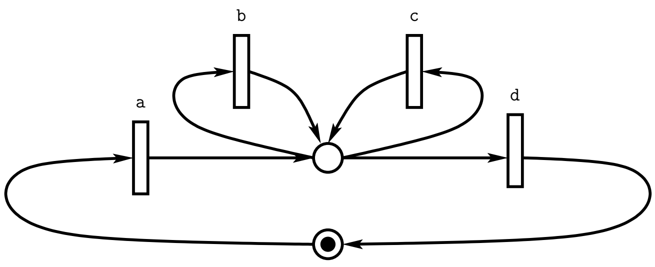


Figure 1: A component interface with services a,b,c and d

Note that a simple loop, using an unlabelled transition, can be introduced in the interface net at the initially marked place. This cycle can represent ancillary processing done by the component that is not directly related to the interface interaction. This processing may also involve the component's eventual termination, if desired.

In any software system, there will naturally be many components and each component can have several interfaces. In order to represent communication between components, the interfaces are divided into *provider* interfaces (*p-interfaces*) and *requester* interfaces (*r-interfaces*) [9].[1]

In the context of a provider interface, a labelled transition can be thought of as a *service* provided by that component. Labelled transitions on the provider essentially denote an entry point into the component. It should be noted that it is possible to have unlabelled transitions on an interface (denoted by ε in the labelling function ℓ_i above). Such transitions may be needed to implement behavioural logic of the interface and do not actually constitute a service.

It is assumed that each service in each p-interface has exactly one labelled representation, so as to prevent ambiguity in the interaction of the component interfaces:

$$\forall t_i, t_j \in T : \ell(t_i) = \ell(t_j) \Rightarrow t_i = t_j.$$

The label assigned to a transition represents a service or some unit of behaviour. For example, the label could conceivably represent a conventional function or method call. The return type and parameters are all encapsulated or abstracted by the label and are of no concern to the model as a whole. It will be assumed that if the p-interface requires parameters from the r-interface, then the appropriate number and types parameters will be delivered by the r-interface. Similarly, it will be assumed that the p-interface will generate an appropriate return value to the r-interface, if required.

Another assumption is that if an r-interface requests any arbitrary service a of a provider component that supports that particular service via its p-interface, then the provider component will be able to satisfy that service (*i.e.* the component servicing the request will not fail due to lack of resources or software faults, for example).

2.2 Interface Languages

Some proposals have restricted interface behaviour by basing the protocol on regular languages, or modest variations thereof [10]. However, by employing Petri nets, this model allows for significant flexibility in the protocol language between components, indeed

[1]Note that this model does not prevent a component from having a provider interact with a requester interface belonging to the same component. This would be an example of a recursive or *feedback* component.

even permitting interface languages recognizable by Turing machines. For example, the protocol languages could conceivably be context-free, which, in the context of modelling the behaviour of a relatively simple data structure, such as a stack, could be quite useful.

Possible sequences of services provided by a p-interface are determined by firing sequences in the Petri net model of an interface, $\mathcal{M}_i = (P_i, T_i, A_i, S_i, \ell_i, m_i)$. That is to say, $\sigma = t_{i_1} t_{i_2} \dots t_{i_k}$ is an initial firing sequence in $\mathcal{M}_i$ iff there exists a sequence of markings $m_{i_0}, m_{i_1}, \dots, m_{i_k}$ such that t_{i_ℓ} is enabled (or fireable) by $m_{i_{\ell-1}}$, m_{i_ℓ} is obtained by firing t_{i_ℓ} in $m_{i_{\ell-1}}$ for $i = 1, 2, \dots, k$, and $m_{i_0} = m_i$ is the initial marking function of $\mathcal{M}_i$. The set of all initial firing sequences of $\mathcal{M}_i$ is denoted by $\mathcal{F}(\mathcal{M}_i)$. Formally, the (initial) firing sequence σ is defined as:

$$\sigma = t_{i_1} t_{i_2} \dots\ t_{i_k} \Leftrightarrow m_{i_0} = m_i\ \wedge\ (\forall\ 0 < j \leq k : t_{i_j} \in E(m_{i_{j-1}})\ \wedge\ m_{i_{j-1}} \overset{t_{i_j}}{\to} m_{i_j}),$$

where $E(m)$ is the set of transitions enabled by m.

Finally, the language of $\mathcal{M}_i$, denoted by $\mathcal{L}(\mathcal{M}_i)$, is the set of all strings over S_i obtained by labelling complete initial firing sequences:

$$\mathcal{L}(\mathcal{M}_i) = \{\ \ell(\sigma) \mid \sigma \in F(\mathcal{M}_i)\ \wedge\ \ell(\sigma) \text{ is a complete sequence of operations } \}$$

where $\ell(t_{i_1} \dots t_{i_k}) = \ell(t_{i_1}) \dots \ell(t_{i_k})$. A *complete sequence of operations* represents a firing of all relevant transitions involved in a provider/requester interaction. As an example, the language associated with the behaviour of the interface presented in Figure 1 is (a(b|c)*d)*. In this case, a complete sequence in a provider/requester interaction would be any repetition of a sequence which started with a and ended with d and had any number of intervening b or c operations.

3 Component Composition and Compatibility

Component composition and compatibility assessment using Petri net models is well established in the literature [11, 12]. Related to this area is the composition and interoperability of web services [13] and verification of workflow composition [14]. While the method presented herein shares concepts with those presented in the literature, especially with respect to the deadlock-free nature of the composition of compatible nets, this paper proposes another method of composition and compatibility assessment that is fundamentally different from those proposed by earlier efforts. In particular, the composition strategy is based on sharing the labels rather than elements of net models, so the interface is composed of services rather than messages or message channels. Interface compatibility is determined by studying the languages generated by the labelled transitions of the provider and requester Petri net interface.

Compatibility of two components is dictated primarily by the behaviour at their respective interfaces. For two components to interact, the provided and requested services must be compatible with one another. This means that not only must all the services required by the requester be made available by the provider, but that the sequence of services that the requester demands must be compatible with the sequence that the provider imposes upon the services being invoked.

This leads to the following definition of *compatible interfaces*: The interface models of requester $\mathcal{M}_i$ and provider $\mathcal{M}_j$ are compatible iff $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$. Observe that this definition implies that the alphabet S_j must be a superset of S_i, $S_i \subseteq S_j$.

3.1 Composition of Interfaces

Informally, the composition can be modelled by "melding" the r-interface, $\mathcal{M}_i$, and p-interface, $\mathcal{M}_j$, together into a single Petri net $\mathcal{M}_{ij} = (P_{ij}, T_{ij}, A_{ij}, S_i, \ell_{ij}, m_{ij})$, assuming $P_i \cap P_j = T_i \cap T_j = \emptyset$.[2] The composition is denoted by $\mathcal{M}_i \rhd \mathcal{M}_j$ with an r-interface as the left-hand argument and a p-interface as the right-hand argument.

The definition of $\mathcal{M}_{ij}$ is based on those transitions in the p-interface and r-interface that have non-empty labels. Let:

$$\begin{aligned} \hat{T}_i &= \{ t \in T_i : \ell_i(t) \neq \varepsilon \}, \\ \hat{T}_j &= \{ t \in T_j : \ell_j(t) \neq \varepsilon \}. \end{aligned}$$

In the composed net, in addition to the existing places in the two interfaces, three more places are added for each requested/provided service. One place is added to the r-interface domain of the resulting net and two places are added to the p-interface domain of the combined net. The purpose of these three places is to act as synchronization points between the requester and provider:

$$P_{ij} = P_i \cup P_j \cup \{ p_{t_i} : t_i \in \hat{T}_i \} \cup \{ p'_{t_j}, p''_{t_j} : t_j \in \hat{T}_j \}.$$

The pairs of places added to the p-interface limit the number of additional places introduced during composition when multiple r-interfaces are allowed to interact with a single p-interface.

With respect to the transitions, all those transitions in the r-interface that have non-empty labels are replaced by a pair of transitions which envelop the additional place introduced in the r-interface domain above:

$$T_{ij} = T_i \cup T_j - \hat{T}_i \cup \{ t'_i, t''_i : t_i \in \hat{T}_i \}.$$

The new places are connected to the transitions as shown in Figure 2.

$$\begin{aligned} A_{ij} = {} & A_i \cup A_j - P_i \times \hat{T}_i - \hat{T}_i \times P_i \cup \\ & \{ (p_i, t'_i), (t'_i, p_{t_i}), (p_{t_i}, t''_i), (t''_i, p_k), (t'_i, p'_{t_j}), (p'_{t_j}, t_j), (t_j, p''_{t_j}), (p''_{t_j}, t''_i) : \\ & t_i \in \hat{T}_i \ \wedge \ t_j \in \hat{T}_j \ \wedge \ \ell_i(t_i) = \ell_j(t_j) \ \wedge \ (p_i, t_i) \in A_i \ \wedge \ (t_i, p_k) \in A_i \}. \end{aligned}$$

The labelling function of the composed net is defined by combining the labelling functions of the two interfaces:

$$\forall t \in T_{ij} : \ell_{ij}(t) = \begin{cases} \ell_i(t), & \text{if } t \in T_i, \\ \ell_j(t), & \text{if } t \in T_j, \\ \varepsilon, & \text{otherwise.} \end{cases}$$

Finally, the composite net has the following marking function which is based upon the markings of the interface nets of the underlying pair of interacting components:

$$\forall p \in P_{ij} : m_{ij}(p) = \begin{cases} m_i(p), & \text{if } p \in P_i, \\ m_j(p), & \text{if } p \in P_j, \\ 0, & \text{otherwise.} \end{cases}$$

Note that the composition technique described in this section precludes the possibility of value failures since all parameter types and return values associated with each

[2]It should be observed that the condition $i \neq j$ is not needed because an r-interface cannot be a p-interface.

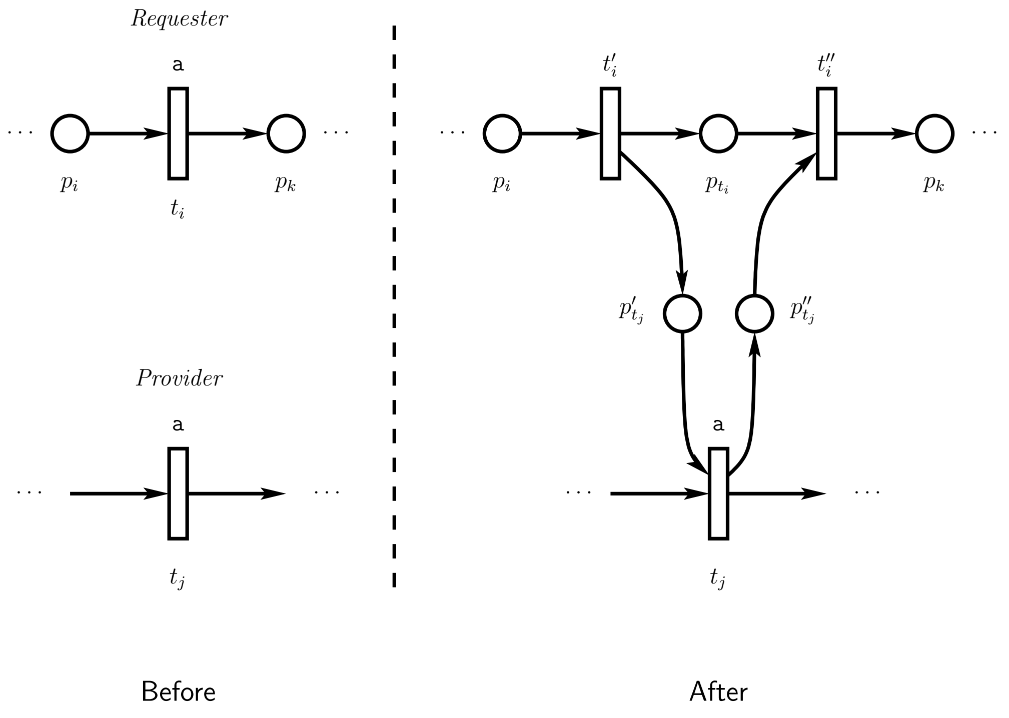


Figure 2: Composing component interfaces

service are abstracted by the transition label — the model assumes that traditional static checking has been performed and that no errors of this nature are possible. Omission failures are successfully identified because if a service is required by the requester, then the corresponding service must be offered by the provider, otherwise the two nets can be immediately deemed to be incompatible. The current model cannot, unfortunately, handle timing failures, but this may be addressed in the future by employing timed Petri nets.

3.2 Compatibility Verification

Interface compatibility means that each sequence of service requests (from an r-interface) is matched by a sequence of identical services in the corresponding p-interface.

Corollary 1
The language of the composition of two interfaces with the same alphabet S, an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$, $\mathcal{M}_i \rhd \mathcal{M}_j$, is the intersection of $\mathcal{L}(\mathcal{M}_i)$ and $\mathcal{L}(\mathcal{M}_j)$,

$$\mathcal{L}(\mathcal{M}_i \rhd \mathcal{M}_j) = \mathcal{L}(\mathcal{M}_i) \cap \mathcal{L}(\mathcal{M}_j).$$

The corollary is a straightforward consequence of the definition of interface composition. In particular, note that the composition technique leaves the structure of both interfaces essentially intact. The primary difference is that token firing is interleaved over the sequence of services of each interface. Ultimately, within the isolated context of each interface, the flow relation remains undisturbed as a result of the composition. Therefore any string generated by the resulting composition can also be generated by each interface. Corollary 1 is supported by the following corollary:

Corollary 2
Let $\mathcal{M}_{ij} = \mathcal{M}_i \rhd \mathcal{M}_j$ and $S_i = S_j$. Each firing sequence σ_i in $\mathcal{M}_i$ such that $\ell(\sigma_i) \in \mathcal{L}(\mathcal{M}_{ij})$ corresponds to a firing sequence $\hat{\sigma_i}$ in $\mathcal{M}_{ij}$ which is obtained from $\sigma_i = t_{i_1}t_{i_2} \dots t_{i_k}$ by replacing each occurrence of each transition t_i such that $\ell(t_i) \neq \varepsilon$ by a triple of transitions t'_i, t_i and t''_i where $\ell(t'_i) = \ell(t''_i) = \varepsilon$, so $\ell(\sigma_i) = \ell(\hat{\sigma_i})$.

This corollary is another consequence of the definition of interface composition, as shown in Figure 2.

Theorem 1
Two deadlock-free interfaces with the same alphabet S, an r-interface $\mathcal{M}_i$ and a p-interface $\mathcal{M}_j$ are incompatible iff the composition $\mathcal{M}_{ij}$ = $\mathcal{M}_i \rhd \mathcal{M}_j$ contains a deadlock.

Proof:
An r-interface $\mathcal{M}_i$ is incompatible with a p-interface $\mathcal{M}_j$ if $\mathcal{L}(\mathcal{M}_i) \supset \mathcal{L}(\mathcal{M}_j)$.

1. $\mathcal{L}(\mathcal{M}_i) \supset \mathcal{L}(\mathcal{M}_j) \Rightarrow \mathcal{M}_{ij}$ contains a deadlock.

   If $\mathcal{L}(\mathcal{M}_i) \supset \mathcal{L}(\mathcal{M}_j)$, there exists an initial firing sequence $\hat{\sigma}$ in $\mathcal{M}_i$ such that $\ell(\hat{\sigma}) \notin \mathcal{L}(\mathcal{M}_{ij})$. The sequence $\hat{\sigma}$ cannot be an initial firing sequence in $\mathcal{M}_{ij}$, which means that it must contain a transition t_k which is not enabled in $\mathcal{M}_{ij}$, so $\hat{\sigma}$ must lead to a deadlock in $\mathcal{M}_{ij}$.

2. $\mathcal{M}_{ij}$ contains a deadlock $\Rightarrow \mathcal{L}(\mathcal{M}_i) \supset \mathcal{L}(\mathcal{M}_j)$.

   By contradiction. The claim is not true, so $\mathcal{M}_{ij} = \mathcal{M}_i \rhd \mathcal{M}_j$ contains a deadlock and $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$. If $\mathcal{M}_{ij}$ contains a deadlock, then there exists a finite firing sequence $\sigma = t_{i_1}t_{i_2} \dots t_{i_k}$ such that $E(m_k) = \emptyset$ where $m_i \overset{\sigma}{\mapsto} m_k$. However, in $\mathcal{M}_i$, σ can be continued ($\mathcal{M}_i$ does not contain a deadlock), so the deadlock can be due only to composition with $\mathcal{M}_j$, *i.e.*, $\ell(\sigma) \notin \mathcal{L}(\mathcal{M}_j)$. This however contradicts the assumption $\mathcal{L}(\mathcal{M}_i) \subseteq \mathcal{L}(\mathcal{M}_j)$.

In summary, the issue of component interface compatibility can be reduced to a problem of detecting deadlocks in a net that results from the composition of two interfaces. It is believed that this model can be extended to handle several requesters interacting with a single provider, possibly through the introduction of coloured Petri nets. Needless to say, as the number of interacting components increases, the problem of deadlock detection in the resulting net becomes increasingly difficult. This challenge may be mitigated by adopting an incremental approach towards interface composition.

4 Examples

This section provides some examples which demonstrate the composition of provider and requester component interfaces using the construction technique presented in the previous sections. To demonstrate the concepts, the example of a database client interacting with a database server will be used. The examples will also highlight the importance of being able to represent interface protocols whose languages are not regular.

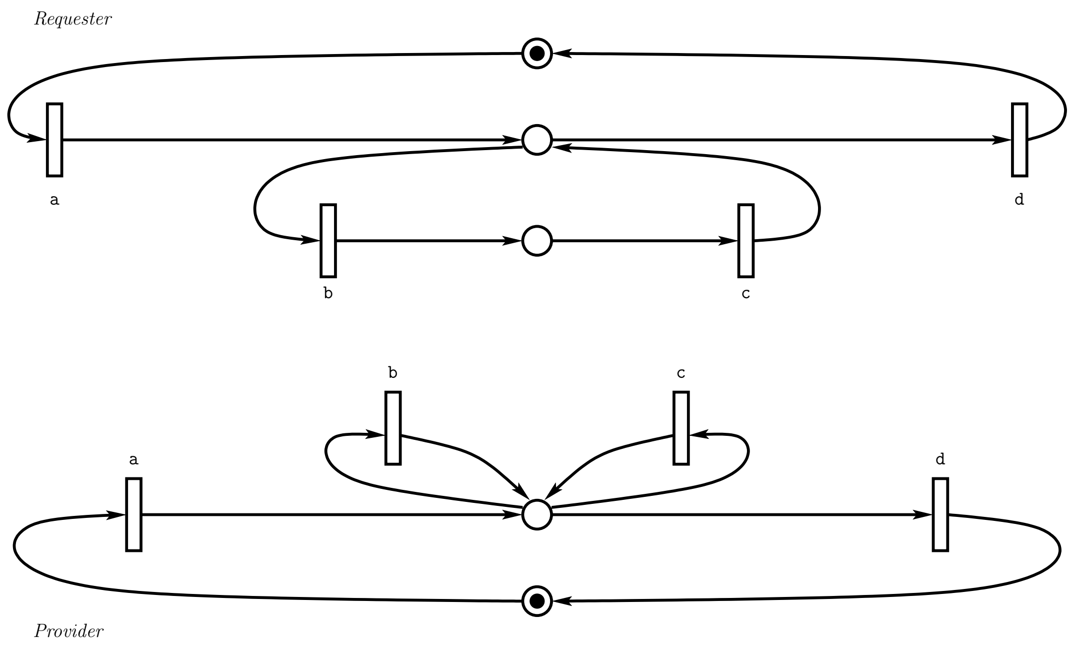


Figure 3: Database requester and provider interfaces

4.1 Database Transactions

As a simple example of the composition of a requester and provider interface modelled as Petri nets, consider Figure 3 which represents a simple database client (requester) and a database server (provider).

The first operation or requested service is denoted by a which could represent a service that opens the database and prepares it for queries, for example. The interface then requests a sequence of operations in which each operation b is followed by a corresponding operation c (these could represent *read* and *write* operations to the database, respectively). Finally, the requester invokes service d which could represent the closing of the database. The behaviour of the requester interface can therefore be represented by the regular language $(\texttt{a}(\texttt{bc})^*\texttt{d})^*$. The provider interface, which represents the database server, imposes the restriction that the a service must be invoked first followed by any sequence of b and/or c services, followed finally by the d service. The behaviour of the database server is therefore denoted by the language $(\texttt{a}(\texttt{b}|\texttt{c})^*\texttt{d})^*$.

The composition of interfaces shown in Figure 3 results in the net shown in Figure 4. This composition is achieved by using the construction technique presented in Section 3.1.

It should be noted that the composition given in Figure 4 enforces the restriction that the database must be opened by the client (requester) before any operations take place upon the database server (provider). Similarly, the requester must close the database in order to satisfy the constraints of the provider. In other words, the net resulting from the composition will be deadlocked if the requester did not perform the a operation first and the d operation last, thereby implying that under such circumstances, the two components would be incompatible.

As another example of a deadlock situation, consider the case where the p-interface and r-interface from the previous example are swapped, so the language of the requester is $(\texttt{a}(\texttt{b}|\texttt{c})^*\texttt{d})^*$ and the provider's language is $(\texttt{a}(\texttt{bc})^*\texttt{d})^*$, and the two nets are recomposed.

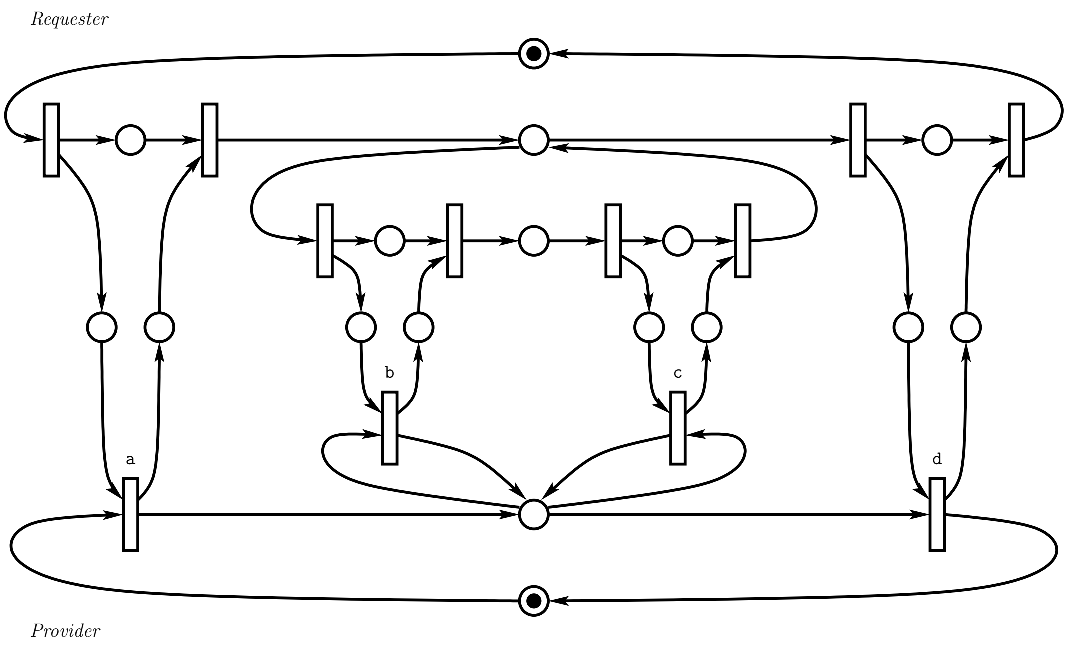


Figure 4: Composition of database requester and provider interfaces

The resulting net would exhibit deadlock as demonstrated by the composition shown in Figure 5. This results in a deadlock situation when the requester invokes service c immediately after invoking a but the provider requires that service b be invoked before service c can be requested. This deadlock demonstrates incompatibility between the two interfaces. In this case, the language of the requester is a superset of the language of the provider.

4.2 Database with Nested Transactions

The example discussed in the previous subsection models a "flat-transaction" system in which *open* and *close* pairs do not nest. Client interaction with a database component that supports nested transactions can also be represented by the model. This highlights the importance of a model being able to represent the context-free nature of the interaction between the client and server as each "opening" of a nested transaction must be matched against a corresponding "closing" of the transaction, which is a behaviour that cannot be modelled with a regular language.

A requester and provider interface that employ nested database transactions can be represented by the Petri nets given in Figure 6.[3] The provider interface keeps track of the number of opened transactions by accumulating a corresponding number of tokens in its top-most place. Similarly, the number of tokens in the bottom-most place of the requester indicate how many transactions have been opened.

These two interfaces can then be composed by applying the composition strategy described in Section 3.1. The resulting net, shown in Figure 7 does not exhibit any deadlock, therefore implying that the the interfaces are indeed compatible.

[3]Note that in the figure, the requester Petri net prohibits the opening of a new transaction in between the b service and c service. This can be easily rectified by introducing a new arc from the b transition to the top-most place in the requester

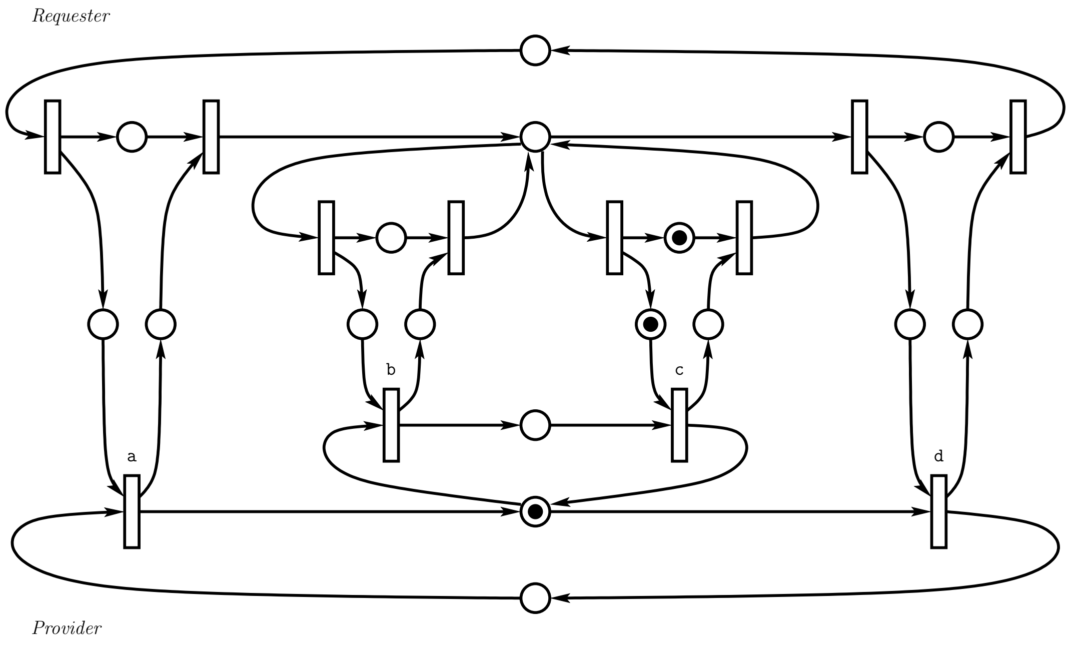


Figure 5: Composition of database requester and provider interfaces resulting in deadlock

Interfaces whose behaviours are much more complicated can also be represented by the model. Indeed, with the introduction of inhibitor arcs in the Petri net of an interface, any interface protocol whose language is recognizable by a Turing machine can be modelled by this approach.

5 Applications and Future Work

Currently, work is underway to design and implement software tools to model the interfaces of generic components. These utilities could conceivably be used to model the dynamic interactions of several components in a theoretical software system. The viability of the interface net model in the context of a software system could be tested using such tools.

Many conventional applications are not overly time dependent with respect the interactions amongst components and modest latencies between component interaction (whether due to hardware or network limitations) are usually acceptable. However, in the case of embedded real-time systems, timing issues are of paramount importance. The model proposed above is insufficient in determining temporal compatibility of two components. Fortunately, through the use of timed Petri nets [15], such timing aspects can be easily added to the model.

Hierarchical composition of components may be represented by the proposed model by constructing a hierarchy of Petri net interfaces [16]. Instead of acting as peers, interfaces can serve as mediators between different levels of a software hierarchy leading to both vertical and horizontal communication within a deployed component-based architecture during compatibility checking. The model presented in this paper is also amenable to establishing the feasibility of replacing an existing component with a new component in a hierarchy. This substitutability aspect can serve to make the upgrading of

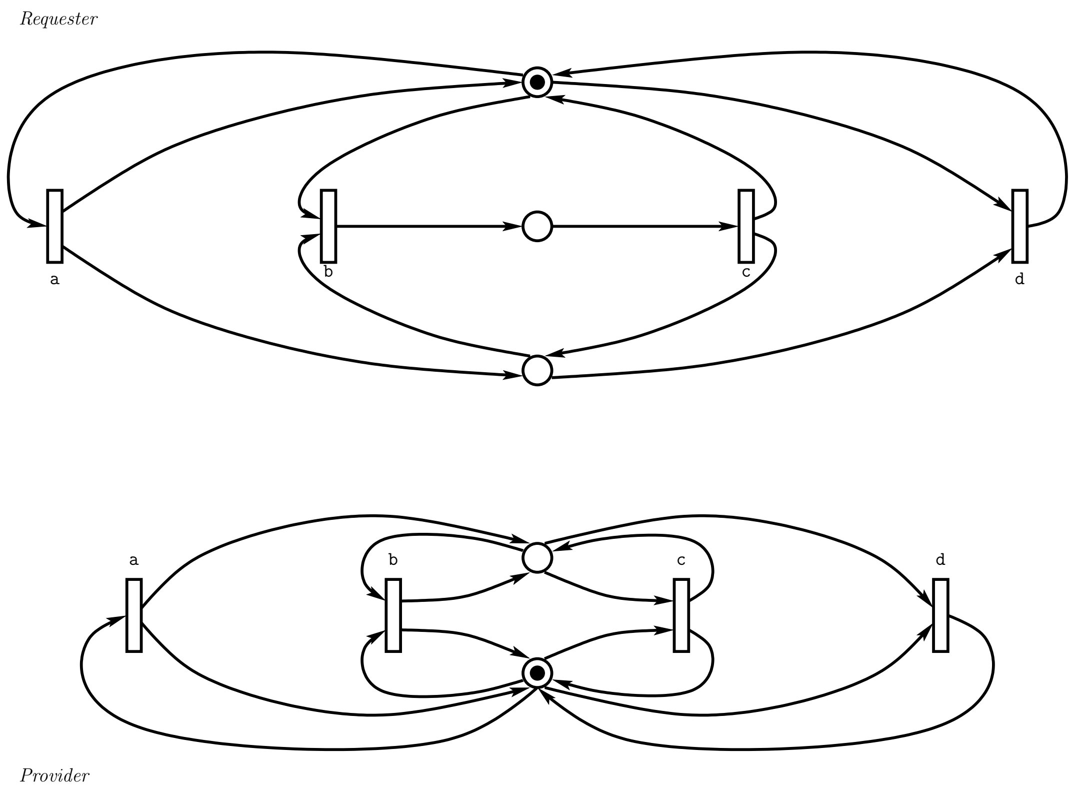


Figure 6: Database requester and provider interfaces using nested transactions

existing systems easier and may also help promote reuse in a software architecture [17].

One issue not fully addressed by this paper is how can one construct the Petri net for an interface when given the corresponding code that implements the interface? Static analysis of the code can, at the very least enumerate the services provided or requested by a component's interface. Static analysis may also reveal, to a limited degree, the sequence of service invocations. However, to accurately determine the complete set of sequences in which the services occur, a dynamic approach must be taken during which all branches of execution must be exercised before a complete Petri net can be deduced.

Another important pragmatic concern that has yet to be resolved is *when* should the compatibility check occur. If the compatibility check can be deferred as late as when the component is deployed into a running environment, then this could allow for the possibility of a software architecture that can dynamically reconfigure itself, potentially giving rise to autonomous, self-assembling software systems that exhibit behaviours that are consistent with a formal requirements specification. Naturally, issues related to state transfer from an old component to a new component would have to be addressed before this possibility can become a reality.

6 Concluding Remarks

Determining the degree to which components are compatible with one another is a multi-faceted problem that, in the general case, requires a comprehensive understanding of both the static and dynamic nature of the components involved. However, by abstracting away the internal, low-level behaviour of components and concentrating solely upon

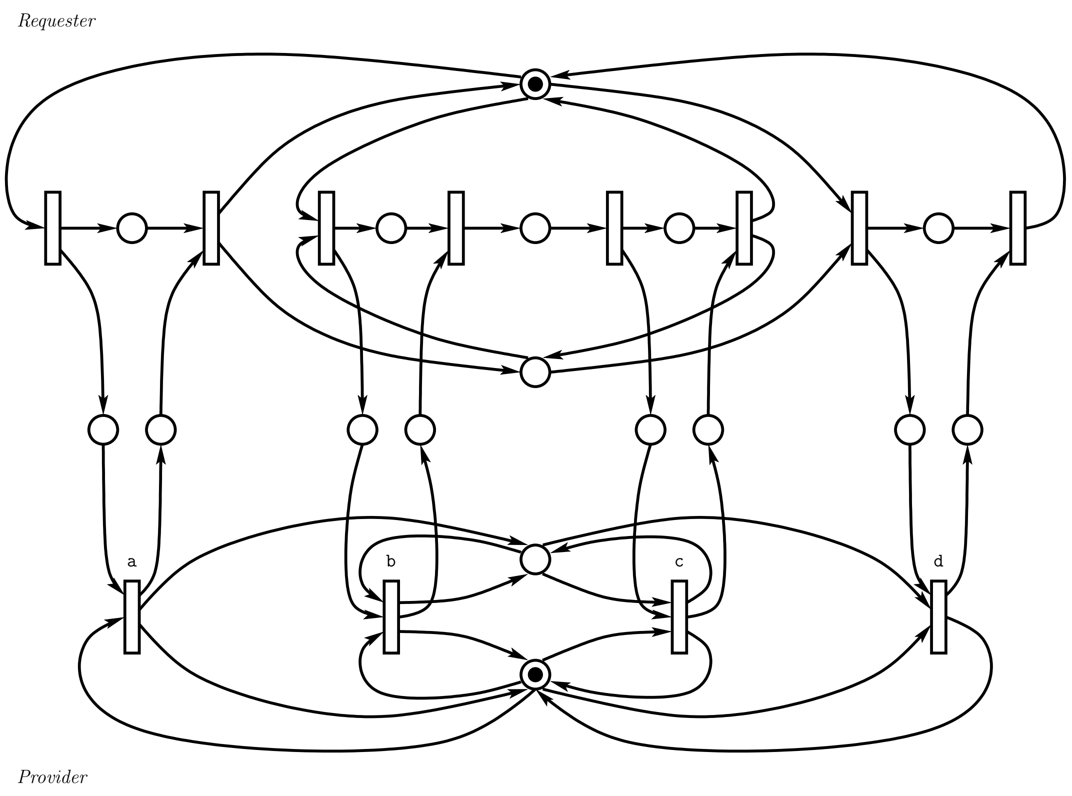


Figure 7: Composition of database requester and provider interfaces using nested transactions

the static and dynamic nature exhibited at their respective interfaces, one can establish whether or not the two components will be able to communicate effectively. This paper presents a formal strategy for composing two components by integrating the Petri nets that represent their interfaces into a single net. If the resulting net does not exhibit deadlock, then the two components are compatible and can function effectively together.

There are several approaches to deadlock detection in Petri nets. The most general one is based on exhaustive exploration of the state space; deadlock states are states which have no *next states*, so they are easily identifiable, but the generation of the entire state space may exhibit the so-called state-explosion phenomenon, *i.e.* the number of states can be an exponential function of some model parameters (*e.g.* the number of interfaces). More efficient methods identify deadlocks on the basis of structural properties and, in particular, siphons and traps [18, 19]. Models of interfaces presented in this paper are composed of multiple cyclic subnets, it is thus expected that structural methods can be used for deadlock detection in this context.

Establishing a well defined and formal method for determining the extent to which two components are able to successfully interact will serve to significantly enhance reuse of software components in a given software architecture and can contribute to the reliable evolution of a deployed component-based software system.

Acknowledgement

The authors are thankful for a number of interesting remarks and comments provided by four anonymous reviewers.

Partial support from the *Natural Sciences and Engineering Research Council of Canada*, through grant RGPIN 8222, is gratefully acknowledged.

References

[1] I. Sommerville. *Software Engineering, 6th edition*. Addison-Wesley, 2001.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications, 2nd edition*. Benjamin/Cummings Publishing Company, Inc., 1994.

[3] N. Medvidovic and R.N. Taylor. A framework for classifying and comparing architecture description languages. In *Software Engineering — ESEC/FSE '97*, volume 1301 of *Lecture Notes in Computer Science*, pages 60–76. Springer-Verlag, 1997.

[4] G.T. Heineman and W.T. Council. *Component Based Software Engineering: Putting the Pieces together*. Addison-Wesley, 2001.

[5] W. Reisig. *Petri-Nets: An Introduction*. Springer-Verlag, 1985.

[6] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[7] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Addison-Wesley, 2002.

[8] W.M.P van der Aalst, K.M. van Hee, and R.A. van der Toorn. Component-based software architectures: A framework based on inheritance of behaviour. *Science of Computer Programming*, 42(2–3):129–171, Feb/Mar 2002.

[9] S. Moschoyiannis and M.W. Shields. Component-based design: Towards guided composition. In Johan Lilius, Felice Balarin, and Ricardo J. Machado, editors, *Proceedings of Third International Conference on Application of Concurrency to System Design*, pages 112–131. IEEE Computer Society, Jun 2003.

[10] R.H. Reussner. Enhanced component interfaces to support dynamic adaption and extension. In Ralph H. Sprague, editor, *Proceedings of the 34th Hawaii International Conference on System Sciences*, pages 1–9. IEEE Computer Society, 2001.

[11] C. Sibertin-Blanc. A compositional partial order semantics for petri net components. In M.A. Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 377–396. Springer-Verlag, 1993.

[12] E. Kindler. A compositional partial order semantics for petri net components. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 235–252. Springer-Verlag, 1997.

[13] A. Martens. Usability of web services. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering Workshops*, pages 182–190. IEEE Computer Society, 2003.

[14] W.M.P van der Aalst. Workflow verification: Finding control-flow errors using petri-net-based techniques. In W. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empircal Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 161–183. Springer-Verlag, 2000.

[15] W.M. Zuberek. Petri nets and timed petri nets in modeling and analysis of concurrent systems, November 2003. Faculty Research Forum — 25th Anniversary of the Department of Computer Science at Memorial University, St. John's, Newfoundland and Labrador, Canada A1B 3X5.

[16] R. Fehling. A concept of hierarchical petri nets with building blocks. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 148–169. Springer-Verlag, 1993.

[17] D. Garland, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17—26, Nov 1995.

[18] Z. Li and M. Zhou. Elementary siphons of petri nets and their application to deadlock prevention in flexible manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics — Part A: Systems and Humans*, 34(1):38–51, Jan 2004.

[19] J. Ezpeleta, J.M. Couvreur, and M. Silva. A new technique for finding a generating family of siphons, traps and st-components: Application to colored petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 1993*, volume 674 of *Lecture Notes in Computer Science*, pages 126–147. Springer-Verlag, 1993.

Towards an Aspect-Oriented approach of Multi-Agent Programming

M. Amiguet[*], A. Nagy[†], J. Baez[‡].

Abstract

Programming large systems requires to cut them in smaller pieces (modules, objects, components, agents, ...). In all but trivial cases, choices must be made about what will be the pieces, and some aspects are inevitably spread out over several parts. In Object-Oriented Programming, these have been identified as "Crosscutting Concerns", and Aspects-Oriented Programming is about a re-unification of these transversal aspects of the system. In multi-agent programming, one of these crosscutting concerns is interaction.

In this paper, we present a multi-agent model that allows to design agents and interaction patterns independently, as first-order objects, and combine them at runtime. The result is a two-dimensional, orthogonal approach of multi-agent systems.

This model features a very flexible agent architecture due to its componential nature. It is targeted towards independance of agents and of interaction patterns, hence increasing their reusability. The model is formalized in an Object-Z/Statechart mixed formalism, and fully implemented in Java.

1 Introduction

Programming large systems requires to cut them in smaller pieces (modules, objects, components, agents, ...). The aims of such a decomposition are, now, obvious. However, in all but trivial cases, choices must be made about what will be the pieces, and some aspects are inevitably spread out over several parts.

In Object-Oriented Programming (OOP), these have been identified as "Crosscutting Concerns", and Aspect-Oriented Programming (AsOP) is about a re-unification of these transversal aspects of the system to facilitate its build, maintenance, customizability and evolution. While the tendency in OOP is to find similarity among classes and push it up in the inheritance tree, AsOP attempts to realize scattered concerns as first-class elements, eject them horizontally from the object structure and then offer a way to compose them.

In Agent-Oriented Programming (AgOP), the primary structuration is evidently the separation into agents. In this case, important crosscutting concerns include interactions. An "aspect-oriented multi-agent programming" would therefore include the possibility to design interaction patterns separately from the agents and combine them afterwards (at design or run time).

Although much work has been done about communication or interaction patterns in Multi-Agent Systems (MAS), few approaches really allow the programmer to do this. In

[*]École d'ingénieurs Arc - University of applied sciences, Le Locle, Switzerland, matthieu.amiguet@bluewin.ch

[†]Institut fur Informatik, Humboldt Universität zu Berlin, Germany, adina_nagy@yahoo.fr

[‡]LIRMM, UMR 5506, Université Montpellier II, France, baez@lirmm.fr

this paper, we will present an approach called MOCA[1] which is probably the closest existing approach to an “aspect-oriented multi-agent programming” paradigm.

The rest of this article is organized as follows. Next section discusses the state of the art of AsOP. Section 3 exposes the main concepts of our architecture and how they are related. Section 5 introduces related works. And finally section 6 discusses the strength and weakness of our approach, together with research perspectives.

2 Aspect Oriented Programming

2.1 General Presentation

In software engineering, architecture and design, the *separation of concerns* became a long-established principle (many AsOP authors refer to the founding principles expressed by Parnas and Djikstra) and many techniques as role and pattern-oriented programming, subject oriented programming, feature-oriented development, co-exist. Among the separation of concerns directions, the increased interest in Aspect-Oriented Programming techniques appeared from the remark that important concerns of modern applications are not easily expressed in a modular way, but often split all-over the code. Aspect-oriented software development was proposed in order to improve the separation of concerns all along the life-cycle of software development.

According to IEEE, a concern is "a set of interests which pertain to the system's development, its operations and any other aspects that are critical or otherwise important to any of its stackholders". Generally, a concern can be any property or matter of interest in a software system. A *space of concerns* is understood by (Sutton Jr. and Rouvellou, 2002) as an organised set of concerns with their respective relationships. Such a space of concerns has the following properties:

- it is multi-dimensional, in the sense that multiple concerns of multiple types may apply to one software unit at a given time. Multi-dimensional separation of concerns (MDSOC (Ossher and Tarr, 2000)) allows multiple and distinct decompositions of a system to co-exist as modules, which overlap and carve up the system in various ways (Elrad et al., 2001a).

- it is highly structured: concerns may be organised by multiple (in)dependent relationships and they commonly have a hierarchical or lattice-like organisation.

The present section gives a glimpse of the main issues and challenges in AsOP and sets the basic definitions. Notions as *aspects*, *cross-cutting concerns*, *decomposition*, *modularization*, *join points* and *weaving* are recalled.

Aspect oriented programming deals with a particular type of concerns, the *aspects*. Aspects are properties which cannot be encapsulated in a general software unit or - as defined by (Kiczales et al., 1997) - an aspect is "a modular unit of crosscutting implementation." Actually, it is the focus on cross-cutting concerns that really distinguishes AsOP from other separation of concerns methods. Two concerns cross-cut if the methods related to those concerns - that is methods which contribute to their design, description or implementation - intersect. One of the difficult issues about cross-cutting concerns is to understand what cuts across what.

Identifying, separately expressing various aspects of a system and describing their relationships allow developers to design a system out of orthogonal concerns. AsOP techniques

[1]Note that the name clash with the name of the present workshop is accidental!

rely both on this decomposition and on particular mechanism to compose or weave concerns together in an over-all system.

Orthogonal decompositions are represented as distinct modules, at design time. Aspect-oriented techniques aim to preserve the design modularity, while representing a given design structure into a program structure. A software preserves modularity (i.e. independence of key parameters belonging to distinct modules) if independent parameters in the design are represented by independent constructs in the program. As indicated in (Sullivan et al., 2002) one special case where modularity is not preserved "occurs when each of a set of apparently independent design parameters is represented by a corresponding program construct, but where the representation of some other design parameter is distributed (cuts) across and is merged into the previous program constructs."

(Sullivan et al., 2002) distinguishes between two situations: when a *program structure* can be said to be an aspect, and when a *design parameter* can be said to be an aspect. (In this paper we shall see that both these types of aspects coexist in MOCA.)

Other classifications of cross-cutting concerns are

- related to the level implied: high-level notions as the security or the optimization, or low level notions as caching or buffering;
- functional concerns vs. non-functional (systemic) concerns,
- behaviour concerns vs. structural concerns, etc.

We have seen that the purpose of AsOP is to provide tools for separating components (objects) and aspects (cross-cutting concerns) from each other.

It implies that AsOP deals with two types of first-order elements: components of the structural realization of the system (objects, classes, interfaces, etc), on the one hand, and scattered concerns, whose identification allows them being extracted from the program structure, on the other hand. Behaviour that would have been spread over the final code is finally congealed in a single structure, by aggregation of aspects into cross-cutting concerns.

The separation of objects and aspects requires a mechanism to recombine (weave) them later, into an over-all system. A compiler, called *weaver*, intertwines the objects (describing functions requested for system) and aspects (that describe cross-cutting concerns). The central notion in the weaving process is the one of join points. A *join point* is generally an element of the program's semantics with which aspect elements coordinate; that is a point where a hook is placed to combine objects and aspects. According to the way concerns and objects are woven, we distinguish two types of composition: static, when the composition of concerns is realized at compile time, and dynamic, that is done along the system's execution. MOCA relies on a dynamic composition of concerns.

Among of the main problems to develop an aspect-oriented system (Elrad et al., 2001b) and (Elrad et al., 2001a) mention the following which are relevant for MOCA:

- *modularization*: the way the AsOP system specifies aspects (aspect description), defines the join points and the behaviour at join points, etc.
- *composition* mechanisms (weaving), deciding whether a dominant decomposition exists or whether the system is based on multidimensional separation of concerns, where concerns belonging to various dimensions are treated as equals;
- visibility of aspects and *conflict mechanisms among aspects*
- *implementation mechanisms*, including - among other features - the decision whether compositions are determined statically or dynamically

2.2 Some Important Approaches

2.2.1 Hyper/J

Hyper/J (IBM, 2004) is based on the notion of *multi-dimensional separation of concerns* (MDSOC) (Ossher and Tarr, 2000). Its objective is to permit the encapsulation of arbitrary kind of concerns and the integration of separate ones. This approach allows developers to decompose their software so it encapsulates all relevant kind (dimensions) of concerns simultaneously, without more than those necessary and without one dominating the others. MDSOC also includes composition ability to allow developers to integrate these separate pieces. A concrete realization of MDSOC for Java is the tool called Hyper/J.

Hyper/J allows a developer to compose a collection of separate models, called hyperslices, each encapsulating a concern, by defining and implementing a (partial) class hierarchy appropriate for that concern. The models typically overlap, and might or might not cut one another. Each model can be understood independently. Hyper/J doesn't require a dominating class hierarchy and doesn't make any distinction between *classes* and *aspects*, allowing any hyperslice to extend, adapt, and be integrated with another one. Hyperslices are loosely coupled, since they never refer directly to one another. This point is achieved through abstract methods: an abstract method in an hyperslice has to be concreted in another one. The developer can integrate separate hyperslices in an unique Java class. To do so Hyper/J operates on class files using hypermodule declaration indicating which hyperslices are to be composed to obtain the final program.

2.2.2 Composition Filter

The filters in the composition filters (CF) (Aksit et al., 1992) model can express crosscutting concerns by modular and orthogonal enhancements to objects. Modularity, since filters have well-defined interfaces and are conceptually independent of the implementation of the object. Orthogonality, since filter specifications do not refer to other filters. Since the observable behavior of an object is determined by the messages it receives and sends, filters can express category of concerns such as inheritance and delegation, synchronization, real-time constraints, and interobject protocols. Filters are attached to objects as enhancement. A CF class aggregates zero or more internal classes and composes their behavior using one or more filters, divided in two sets: input and output filters. An internal class may be again a CF class or it may be a standard one.

Whenever a message arrives at the filter, a matching is initiated. The matching process primarily involves the targeted (named object) and the selector of the message. If the matching succeeds, the message is said to be accepted by the filter (and it is sent to the target), if not, the message is said to be rejected. Some filter types are:

- *Dispatch* which sends the message to its target if accepted, if not continues with the next filter;
- *Error* which sends the message to its target if accepted, if not an exception is raised;
- *Meta* which sends the message as parameter of a new one to a new target if accepted, if not continues with the next filter.

2.2.3 AspectJ

In AspectJ (Kiczales et al., 2001), a join point is the point in the execution flow of a program when a method is called, and when method returns. Each method call itself is one join point. Pointcut designators are used to identify a subset of all the join points in the program flow. Programmers can define named pointcut designators, and pointcut designators can identify join points from many different classes, AspectJ also allows specification of a pointcut in terms of properties of methods rather than their exact name. In AspectJ, advice declarations are used to define additional code that runs at join points. *Before advices* run before the method begins running. *After advices* run just after the method has run. *Around advices* run when the join point is reached and has explicit control over whether the method itself is allowed to run at all. Pointcut designators can expose certain values in the execution context at join points and those values can be used in advice declarations. Finally, AspectJ is easy to use but the fact that aspects often cannot be understood without references to their model reduces their reusability.

2.2.4 DJ Library

DJ Library's aim is to allow Java programmers to experiment with AsOP without having to learn an extension of Java or to preprocess source files. This library is based on the idea of adaptive methods (known in (Lieberherr, 1996) as propagation pattern). Such a method encapsulates the behavior of an operation into one place, avoiding to disseminate an operation across several classes, but also abstract over the class structure, avoiding to tangle to much information about the structure of classes. To do this, the behavior is expressed as a high-level description of how to reach the classes participating in an operation (a traversal strategy) and what to do when the participants have been reached (adaptive visitor). The class structure (to traverse) is directly the one used by Java at run-time: the DJ Library exploits the reflection ability of Java to know it. The join points are those defined by the traversal of the structure. Then the class structures, the traversal strategy and the visitors can easily evolve independently. This library is easy to use but only a limited class of behavioral concerns can be expressed as adaptive methods.

2.3 Overview of Recent Works

Although recent works on aspects are often Java-based, many other languages benefit from aspectual extensions: C++, C# and the .NET framework, Smalltalk, Python, Perl and Ruby among others (see (The AOSD Steering Committee, 2004) for details). Works are also done towards an aspect-oriented extension of UML (Clarke and Walker, 2002).

Whatever language is used, one key issue is when and how classes and aspects are "woven". Many approaches do this at compile time; however, other possibilities exist:

- Load time weaving. This can be very useful e.g. in situations where one wants to modify or adapt a third-party library of which one doesn't have the sources (Kniesel et al., 2004).
- Run time weaving (also called *dynamic weaving*). This kind of weaving allows for dynamic program adaptation to runtime circumstances (Schult and Polze, 2003; Popovici et al., 2002)

As a general rule, a later weaving is more flexible, but also entails more overhead and a greater difficulty to ensure correct execution.

Modularity and *reusability* are two keywords of AsOP. As they also are center concepts in component-based approaches, it is no wonder to find several works about aspect-oriented components. (Grundy, 2000) proposes a decomposition in "vertical slices" (components) and "horizontal slices" (aspects) to improve readability and reusability of code. The Dynamic Aspect-Oriented Platform (Pinto et al., 2002) proposes a similar approach including runtime weaving of aspects. QCSS (Sassen et al., 2002) proposes a methodology based on aspects, components and contracts to guarantee functional behaviour, structural properties and synchronization.

Although AsOP is a relatively new area of research, several applications have been proposed in very various fields like prefetching in OS development (Coady et al., 2001), bug tracking (Bodkin, 2003) and Quality of Service (Schantz et al., 2002).

3 MOCA: An Aspect Oriented Multi-Agent Approach

In the context of this workshop, we suppose the reader familiar with general Agent-Oriented programming concepts and we will directly proceed to the presentation of the MOCA model. For the sake of clarity, we will provide a presentation in a traditional multi-agent style and delay the AsOP discussion until the end of the section.

From an Agent-Oriented point of view, MOCA has the following characteristics:

1. It is a dynamic behaviorist organizational model. This means that agents can take and leave roles at runtime, and that taking a role entails to follow a given behavior.
2. Agents have a componential architecture.
3. Agents can take several roles simultaneously, and a role conflicts management mechanism is provided.
4. The model is fully implemented in Java above the MadKit multi-agent platform.

The next section will briefly present MadKit and its underlying model Aalaadin, as well as their relationships to MOCA. Section 3.2, will then present the main concepts of the model; sections 3.3 and 3.4 will go into deeper detail. Sections 3.5 and 3.6 will give an overview of organizational dynamics and role conflicts management. Finally, section 3.7 will discuss links between MOCA and AsOP.

3.1 Aalaadin, MadKit and MOCA

MOCA is based on the Aalaadin methodology (Ferber and Gutknecht, 1998), which includes two conceptual levels: the *concrete* level, which includes the notions of *agent*, *group*, and *role* (only as names for agents in a group), implementing the actual multi-agent structure; the *abstract* level describes valid interactions and structures of groups (which we call organizations). However, the abstract level is methodological: although the concrete level is fully formalized (Ferber and Gutknecht, 1999), the abstract level is never formalized nor implemented. In (Hübner et al., 2002), both levels are formalized but only from a specification perspective.

MadKit (Ferber et al., 2004) is a multi-agent platform based on the Agent-Group-Role model (Ferber and Gutknecht, 1998), on which the Aalaadin methodology is targeted (Ferber and Gutknecht, 1998). The services provided by this platform are of two kinds: (i) The agent services (Management of messages, Agent life cycle and acquaintances); (ii) The group services (group creation, entering and leaving, information).

Note that a group is explicitly not a recursive notion in MadKit in the sense that a group cannot be an agent.

Even if methodologically the groups are thought as instantiations of organizations and roles are seen as instances of behaviour patterns, this is not implemented in the platform. In fact, roles and groups are just names for structuring the MAS architecture. In order to overcome this limitation, we want to introduce explicitly *organization* and *role descriptions* as first order objects of which groups and roles are instantiations. Therefore, our contribution with MOCA is to propose formalizations for representing the concepts of Aalaadin and to enrich MadKit with a reification of the *abstract* level concepts, thus providing a complete, operational formalism to build multi-agent systems in an organizational-centered approach (Lemaître and Excelente, 1998). MOCA is implemented in such a way that any multi-agent platform providing communicating agents and group structures could be used in place of MadKit.

Additionally, to ensure orthogonality and hence improve reusability of organizations and role descriptions, we limit the possible interactions to those that take place inside a given group. The only opportunity for two different groups to exchange and cooperate is when some agents participate in both of them at the same time. This last option is mentioned in the Aalaadin methodology but not currently enforced by MadKit.

3.2 Overview of MOCA Concepts

To describe the organizational structure on the one hand and the multi-agent system on the other, our model is composed of two different levels: the descriptive or organizational level and the executive or multi-agent level. Additionally, we distinguish the external or behaviorist point of view on the system from the internal point of view of the agent. In the external point of view we observe interacting (through influence) agents achieving acquaintance structures or groups whose recurrence can be abstracted away as relational structures forming organizations. In the internal point of view, we describe the agent architectures classified into agent types (Reactive, deliberative, BDI, etc.) and providing competences. Role descriptions and their instantiations articulate both by providing at the same time the recurrent pattern of interactions from an external point of view and the way it is achieved from an internal point of view, using and relying on agent competences. Most probably, separating these two aspects would allow further flexibility but have not been taken into account in MOCA. Figure 1 represents the important concepts for each level and from each point of view and the correspondence between them. Figure 2 describes the links between the internal concepts.

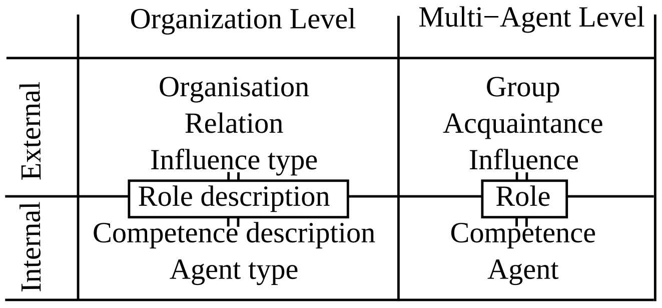


Figure 1: The concepts of MOCA

Most of the notions of the MAS level exist in MadKit, except for the notions of *influence* and *competence* which will be described further in this section. Our major contribution from an AgOP point of view is hence the explicit representation of the Organization level.

Agents in MOCA must have a very flexible architecture because of their possibility to take and leave roles at runtime. To achieve this flexibility, we chose a componential approach. A full description of the componential aspects of MOCA is out of the scope of

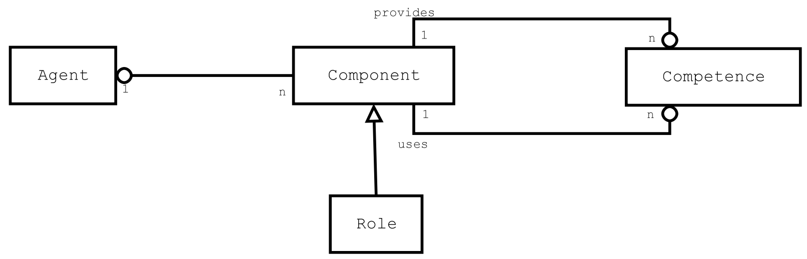


Figure 2: The internal concepts of MOCA

this paper, but this presentation will introduce the minimal concepts needed for a good understanding of the model.

In the next subsections, we will describe in detail the notions of figures 1 and 2.

3.3 The Organization Level

3.3.1 Organizations and Relationships

An organization is defined as a recurrent pattern of interactions from a given point of view (financial exchanges, goods exchanges, ants collective behavior, etc.) and very often attributed with a rationale (ensure equilibrium, feed the ant colony, etc.). In our case, it is formalized as a set of role descriptions and relationships between them. Cardinalities can be specified on both role descriptions and relationships, to describe how many times they can be instantiated as roles and acquaintances, respectively. Figure 3 pictures an example of an organization for the FIPA Contract Net protocol (Collective, 2003). There are two role descriptions in this organization: the initiator and the participant; in a given instantiation of the organization, there can be only one initiator, but of course several participants, each of them being possibly in acquaintance with the initiator.

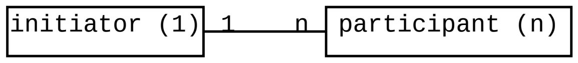


Figure 3: A simple organization for the Contract Net Protocol

3.3.2 Influence Types

The influence type specifies the kind of interaction a role can receive and generate. In order to be general, we use the term influence to cover in the same abstraction forces for physically situated roles, events and up to speech acts (ACL, KQML, etc.) for socially situated roles (Ferber and Müller, 1996).

3.3.3 Role Descriptions

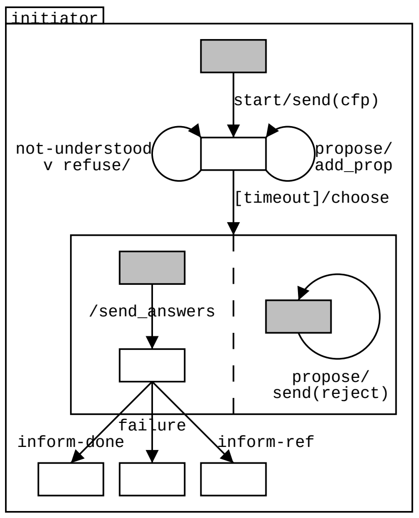
A role is defined as a recurrent behavioral pattern within an organization. The role description represents the way this behavioural pattern is generated, being the internal account of the external pattern. To formalize it, we chose to use a formalism strongly based on (Hilaire, 2000), which is a combination of Statecharts (Harel, 1987) and Object-Z (Duke et al., 1991). The formalism is quite intuitive, so we expose it quickly on an example, without further formal description (for details refer to (Hilaire, 2000)).

Figure 4 pictures a typical role description corresponding to the initiator role of figure 3.

Initiator

$MOCARoleDescription[redef\ \ behavior]$

behavior



$propositions : \mathbb{P}\ Proposition$
$chosen : Proposition$

$req\_competences = \{send, evaluate, timeout\}$
$prov\_competences = \{delegate\_task\}$

delegate_task

$generate\_event(start)$

add_prop

$\Delta propositions$
$prop?$

$propositions' = propositions \cup prop?$

choose

$\Delta chosen$

$\forall\, p \in propositions \bullet evaluate(chosen') \geq evaluate(p)$

send_answers

$\forall\, p \in propositions - \{chosen\} \bullet send(p, reject)$
$send(chosen, accept)$

Figure 4: The *initiator* role description

This description is based on Hilaire's framework (Hilaire, 2000), which can be seen in the inheritance statement of the first line. The heart of the description is of course the statechart specifying the behavior. Statecharts are an extension of finite state automata used in UML, with and-or hierarchy of states; hence the dotted line in figure 4 means that the two halves of the super-state have to run in parallel. Note that we use a slightly non-standard notation for default states, which are pictured as gray boxes.

Transitions are labeled with three-parts expressions of the form *influence[condition]/action*. This means that the arrival of *influence* triggers the firing of the transition, which can take place only if *condition* holds. If the transition is fired, then *action* is executed.

Each part of the expression can be omitted.

Actions can be calls to agent competences (e.g. *send*); the required competences of the agent must be listed (variable *req_ competences*) in order to check if the agent is able to take the role. Usually, at least *send* will be listed there, as it corresponds to the capacity of sending influences to other agents. Our model also allows for internal methods and variables, specified in object-Z notation (e.g. *add_prop*).

An important feature of MOCA roles is that they can also *provide* competences to the agent. Thus endorsing a role can give the agent new abilities, permanently or not. For example, the role described here provides the competence *delegate_ task*, which can be used by another role in the agent to delegate a task to another agent. When this competence is called, the execution of the role is started (through the generation of the *start* event).

From a componential point of view, roles are a particular kind of component, and required and provided competences correspond to its input and output ports, respectively.

3.3.4 Competence Descriptions

The competence descriptions are specifications of the services an agent has to provide in order to be able to play the role; typical examples are the ability to send and to receive messages, to compute an offer or evaluate between offers for a contract net. This notion allows the description of the role at a very abstract level, essentially the management of interactions.

3.4 The Multi-Agent Level

At the multi-agent level we find the instantiation of the various concepts at the organizational level.

3.4.1 Groups

A group is an instance of an organization made of agents playing the various roles of the organization. The only difference from the same notion in MadKit is that this instantiation is made explicit.

3.4.2 Roles

The role in MadKit is just a name in a group; it is not expected that roles with the same name in different groups are linked to the same behavior. In our higher abstraction level, two roles with the same name in two different organizations can produce different behaviors, but two roles with the same name in two different groups instantiating the same organization will have the same behavior. Each role in an agent is the instantiation of the role's state (automaton state and internal variables) and a reference to the role description.

3.4.3 Acquaintances

The acquaintances of an agent are, as usual, the agents it can communicate with. The link between acquaintances and relationships is that an agent can be in acquaintance with another agent if and only if there is at least one relationship between the roles played by the respective agents.

3.4.4 Agents, Influences and Competences

An agent is an autonomous entity able to create new groups by instantiating organizations, to take and to leave roles in existing groups. An agent can play none to many roles at any given time. The typical execution of an agent includes:

1. Get the influences coming from other agents dispatch them on the different roles.
2. Manage components communication (incl. competence calls).
3. Send the influences emitted by its roles to other agents.
4. Control the execution of its components in order to avoid negative interferences.
5. Manage its components (incl. taking and leaving roles).

The three first tasks are more or less straightforward.

The fourth one is more problematic and will be described in section 3.6.

As for the last task, it is out of the scope of MOCA: the very idea of the approach is to provide tools that allow an agent to take and leave roles at runtime, but the reason why he would do so is left to the designer of the system.

3.5 Organizational dynamics

In order to be able to create a group instantiating an organization and to enter and leave it, we need a mechanism to do so. This is clearly a “meta” level with respect to the organizational one, but to preserve conceptual homogeneity of our model, we choose to implement this mechanism as an organization.

The idea is to have a *Managing group* in the system in which all agents take part[2]. The corresponding *Managing organization* has three role descriptions (RD’s) which are the *YellowPages* RD, the *Manager* RD and the *Requester* RD.

We are not going to describe the details of the organizational mechanisms in MOCA (for more details refer to (Amiguet, 2003; Amiguet et al., 2002)), but they allow for group creation and management without any inter-group communication. However, if we want our system to be more than a few groups functioning independently, we have to provide a collaboration mechanism between groups. The way to do this without introducing explicit inter-group communication is to allow agents to have several roles in different groups at the same time.

Hence the local coordination of roles inside the agents achieves groups coordination at the system level. The way roles are coordinated by an agent is of course part of the agent’s autonomy, but in order to ease the agent’s design we provide a generic and powerful implicit communication mechanism: we allow roles to provide competences to the agent. Hence the role executions rely only on agent competences, but the agent has the possibility to use roles to implement some of them. This allows elegant implicit role communication inside the agent, but to be effective, it requires a way of dealing with role conflicts.

3.6 Role Conflicts Management

Whenever roles imply a given behavior, the problem arises of roles conflicts; to cope with that, we introduce the notion of *mutual ignorance*. A system is said to ensure mutual

[2]In a large system, it is possible to have several of them to avoid a bottleneck effect.

ignorance on roles if roles can execute without having to know whether the agent has other roles running.

In MOCA, mutual ignorance is generalized on components execution and is ensured by the following mechanism:

1. Whenever a component A calls a competence of component B, A is registered as a user of B. When A makes its last call to this competence, B will be notified automatically.

2. Component B has the possibility to *accept* or *reject* competence calls. An *acceptance policy* must be provided to determine whether B accepts one or several users, or one "full" user and several "read-only users", etc.

3. If the call is rejected, A will *block* the corresponding transition in its statechart. B will notify A when it is ready again to get calls.

This provides only a very coarse idea of the role conflict mechanism in MOCA (for more details refer to (Amiguet, 2003)). The result is a mutual exclusion on zones which are automatically determined at runtime depending on the participating components and the nature of the involved resource. This mechanism can be proved to ensure mutual ignorance, and this with *minimal exclusion zones* (Amiguet, 2003).

3.7 MOCA and AsOP

The MOCA approach allows the programmer to design separately agents on the one hand and organizations on the other hand, and then combine these elements freely in a running system.

This is clearly a two-dimensional separation of concerns: agents, as the primary decomposition, form "vertical slices" and organizations form "horizontal slices". Orthogonality is ensured by the two characteristics that agents cannot communicate outside of a group and groups cannot communicate without a shared agent.

In other terms, in any MAS, interaction protocols crosscut over agents. MOCA allows them to be represented as unified aspects, called organizations.

This is the main contribution of MOCA towards an Aspect-Oriented Agent Programming. However, the treatment of other (crosscutting) concerns can also be simplified with this approach: see (Ferber and Gutknecht, 1998) for a reflexion about how MadKit-based approaches enhance security and (Amiguet, 2003) for perspectives on formal validation of MOCA-based systems.

From a pure AsOP perspective, MOCA has the outstanding characteristic that it offers runtime weaving of aspects including a mechanism to resolve conflicts.

4 Example of Use

This architecture has been tested on a number of examples among them the famous prey/predator benchmark (Müller et al., 2001) and a foot-and-mouth epidemic simulation. This last example was first proposed by (Durand, 1996), and was restated in a somewhat simplified formulation by (Hilaire, 2000). We have based our experiment on Hilaire's model.

Figure 5 pictures the structure of the system, which consists of three agent types and two organizations. The meaning of the *Stockbreeder* and *Livestock* agent types is straightforward; note that the *Disease* has also been reified as an agent. We now quickly describe the two organizations:

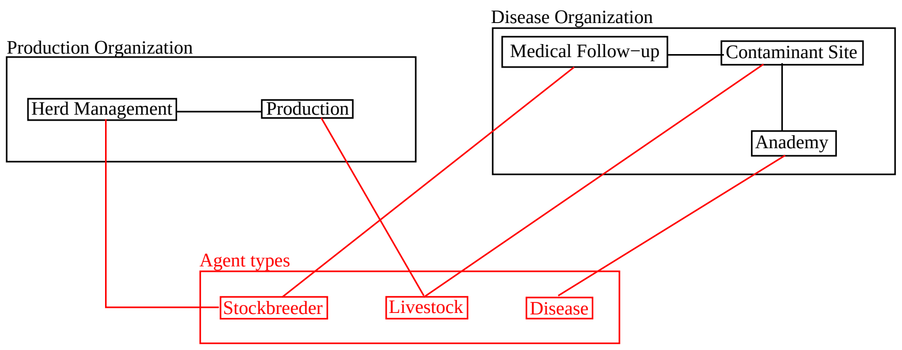


Figure 5: The foot-and-mouth epidemic example

The Production Organization is composed of two roles: the *Herd Management* role is responsible for selling animals, and moving them to grass-land in spring and back to stalls in fall. The *Production* role is responsible for simulating the birth of calves.

The Disease Organization is composed of three roles: the *Medical Follow-up* role is responsible for detecting the presence of the disease and initiating a treatment; The *Contaminant Site* role deals with the effect of the treatment and the *Anademy* role simulates contagion and healing.

A more complete description of this simulation, including the full specification of the roles, can be found in (Hilaire, 2000).

Figure 6 pictures the result of the simulation: the number of sick animals increases in winter, because they are located in stalls, were contamination rate is high; it decreases in summer when the animals are outside. The total number of cows varies as a result of the births and selling. Our results are very similar to those of (Hilaire, 2000) and could be obtained with a low cost in term of development time.

5 Related works

As noted in (Garcia et al., 2002), software engineering approaches for MAS development fall in two categories: (i) agent-based software engineering, and (ii) object-oriented software engineering for agent systems. Both can take advantage of aspect-oriented approaches.

(Garcia et al., 2002) is an example of an aspectual approach of the second category, where aspects are used to modularize agent capacities and properties. Note that the aspects usually don't cut across agents; the approach is therefore quite similar to coarse-grained component-based approaches like Voyelles (Ricordel, 2001).

MOCA falls into the first category. We don't know of any other agent-based SE approach explicitly referring to AsOP; however, several organization-based works bear some similarities with our proposal.

As in MOCA, the focus of (Durand, 1996), Aalaadin (Ferber and Gutknecht, 1998), (Ferber and Gutknecht, 1999), Opera (Dury, 2000), the model of (Hilaire, 2000), MOiSE+ (Hübner et al., 2002), Gaia (Wooldridge et al., 2000), and other related models is on the organizational structure of a multi-agent system, as well as on the study of agents'

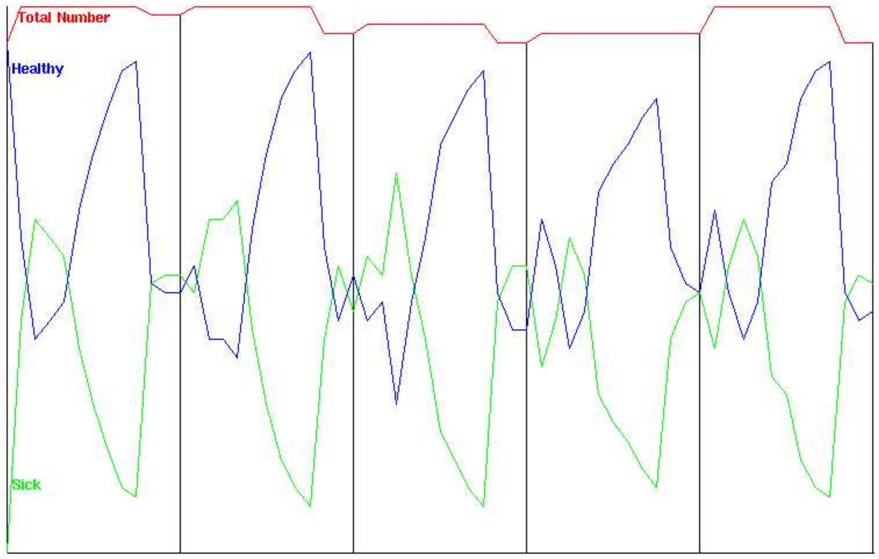


Figure 6: The results of the simulation

behaviour induced by this structure. (Unlike in MOCA, social dynamics is missing in several of these models).

The works on Aalaadin (Gutknecht, 2001) and MadKit (www.madkit.org) are based on the Agent-Group-Role model which allows for a structuration of the MAS in levels and for an organizational dynamics, while considering concerns as security, heterogeneity of agents and reusability. However, roles in Aalaadin are not given a behavioural content and the notions of organisation and group are *abstract concepts*. Cassiopee (Collinot et al., 1996) relies on the expression of both collective behaviours and elementary behaviours of agents. It follows that the behavioural dimension is strongly present, but the model has no structuration in levels and no reified organizational concepts. MOCA inherits equally from Hilaire's Object-Z/statechart model (Hilaire, 2000), employed to represent behaviours associated to roles, which allows for the abstraction and formalization of collective behaviours. While Hilaire's model covers the design and test of organizational structures, it neither covers the conceptual dimension, nor allows for organizational dynamics or conflict management for multiple role endorsement. Opera (Dury, 2000) is built around a specification of interactions, which is revisited within the *Role description* concept of MOCA. However, MOCA allows for richer behaviours associated to roles, and for a finer management of role conflicts within an agent.

In (Wooldridge et al., 2000, and related papers) Gaia represents an organizational methodology for agent-oriented analysis and design, which bears some similarities with MOCA, as it is based onto two abstract models - a “mixed” role model and an interaction model. While Gaia is a valuable methodology for Agent-Oriented Software Engeneering, organizational concepts are present only at design phase while nothing is said about multi-organizational systems and multiple role endorsement.

In MOISE + (Hübner et al., 2002; Hübner, 2003), whose initial purpose was to reconcile organizational and global-plan centered models, three dimensions are used to describe a MAS: the structure expressed through roles and groups, the functioning (global plans and tasks) and deontic relations or other norms (agents' obligations, norms, responsibilities, permissions etc.) The structural dimension is centered on three main concepts - roles, role relations and groups - which are employed to build the individual, social, and collective structural levels of an organization. MOISE + doesn't rely on a behavioural definition of roles, but sees a role as a set of constraints that an agent ought to follow when it accepts to enter a group. The constraints imposed by the role are defined in relation to other roles (in the collective structural level) and in relation to global plans. the social level is used to specify relations between roles, that is links used to constrain agents after they have accepted to play a role. The collective level imposes some constraints regarding the roles an agent can play at the same time. The functional dimension describes the way global goals are decomposed by plans and distributed to the agents by missions. This decomposition supposes, at collective level, a global plan decomposed as a social schema and, at individual level, missions an agent may commit to. Deontic dimension relates the structural and functional dimensions. The functional and deontic dimensions insure an organizational dynamics. Besides the attribution of behaviours to roles, the reification of the notions of organization and role, MOCA relies on other choices different from MOISE +: on a specification of roles in the organizational structure (lack of inheritance) and on the fact that role endorsement is not only multiple but equally a matter of agents' autonomy, as driven by agents' internal goals.

6 Conclusion

In allowing the programmer to design organizations and agents independently, the MOCA model can be seen as an Aspect-Oriented approach of agent-oriented software design. From this point of view, agents form "vertical slices" and groups "horizontal slices" in the MAS. MOCA enforces a complete orthogonality of these slices, thus allowing to develop very reusable modules (both agents and organizations).

For the time being, MOCA has only been tested on relatively small systems (Amiguet, 2003; Amiguet et al., 2002). One important perspective is therefore larger scale testing. As MOCA is planned to be included in the next version of MadKit, this will probably encourage its use on larger-scale systems.

The current implementation, though functional, is more a proof of the concept than a production-ready platform (due, among others, to its heavy use of Java reflection). A reimplementation is in work as a part of the MIMOSA project (Collective, 2004) that should result in a more efficient and user-friendly platform.

Objects, components and agents provide very powerful structuring paradigms for building software systems. However, a stronger structuration also implies a stronger splitting of some concerns over the different parts of the system. It is therefore necessary to explore ways of dealing with these crosscutting concerns to keep them under control. MOCA is an attempt to provide solutions to some of the problems occurring along this way. Even if this model is far from perfect, we think that multi-agent programming could benefit a lot from a deeper understanding of AsOP techniques. Along the way, techniques could be developed (like, MOCA's conflicts management mechanism) that are of interest to the AsOP community.

References

Aksit, M., Bergmans, L., and Vural, S. (1992). An object-oriented language-database integration model: The composition-filters approach. In Madsen, O. L., editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 372–395, Berlin, Heidelberg, New York, Tokyo. Springer-Verlag.

Amiguet, M. (2003). *MOCA: Un modèle componentiel dynamique pour les systèmes multi-agents organisationnels*. PhD thesis, Université de Neuchâtel.

Amiguet, M., Müller, J., Báez-Barranco, J., and Nagy, A. (2002). The moca platform: Simulating the dynamics of social networks. In Sichman, J. S., Bousquet, F., and Davidsson, P., editors, *Multi-Agent-Based Simulation II*, number 2581 in LNAI, pages 70–88. Springer-Verlag.

Bodkin, R. (2003). aTrack project home. `https://atrack.dev.java.net/`.

Clarke, S. and Walker, R. J. (2002). Towards a standard design language for AOSD. In *1st International Conference on Aspect-Oriented Software Development*, Enschede.

Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., and Ong, J. S. (2001). Structuring operating system aspects. *CACM*, 44(10):79–82.

Collective (2003). The foundation for intelligent physical agents (FIPA). `http://www.fipa.org`.

Collective (2004). Le projet MIMOSA. `http://lil.univ-littoral.fr/Mimosa/`.

Collinot, A., Ploix, L., and Drogoul, A. (1996). Application de la méthode cassiopée à l'organisation d'une équipe de robots. In Müller, J.-P. and Quinqueton, J., editors, *JFIADSMA'96*, pages 137–152. Hermes.

Duke, R., King, P., Rose, G., and Smith, G. (1991). The Object-Z specification Language. Technical report, Software Verification Research Center, Departement of Computer Science, University of Queensland, Australia.

Durand, B. (1996). *Simulation multi-agents et épidémiologie opérationnelle*. PhD thesis, Université de Caen.

Dury, A. (2000). *Modélisation des interactions dans les systèmes multi-agents*. PhD thesis, Université Henri Poincaré.

Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., and Ossher, H. (2001a). Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38.

Elrad, T., Filman, R. E., and Bader, A. (2001b). Aspect-oriented programming. *Communications of the ACM*, 44(10):29–32.

Ferber, J. and Gutknecht, O. (1998). A meta-model for the analysis and design of organizations in multi-agent systems. In *ICMAS'98*, pages 128–135. IEEE Computer Society.

Ferber, J. and Gutknecht, O. (1999). Operational semantics of a role-based agent architecture. In *ATAL'99*.

Ferber, J., Gutknecht, O., and Michel, F. (2004). The MadKit project (a multi-agent development kit). http://www.madkit.org.

Ferber, J. and Müller, J.-P. (1996). Influence and reaction: a model of situated multi-agent system. In *ICMAS'96*, Kyoto. AAAI Press.

Garcia, A., Silva, V., Chavez, C., , and Lucena, C. (2002). Engineering multi-agent systems with patterns and aspects. *Journal of the Brazilian Computer Society*.

Grundy, J. (2000). Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 10(6).

Gutknecht, O. (2001). *Proposition d'un modèle organisationnel générique de systèmes multi-agents et examen de ses conséquences formelles, implémentatoires et méthodologiques*. PhD thesis, Université des Sciences et Techniques du Languedoc.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274.

Hilaire, V. (2000). *Vers une approche de spécification, de prototypage et de vérification de systèmes multi-agents*. PhD thesis, Université de Franche-Comté.

Hübner, J. F. (2003). *Um modelo de reorganização de systemas multi-agentas*. PhD thesis, Universidade de São Paulo.

Hübner, J. F., Sichman, J. S., and Boissier, O. (2002). Spécification structurelle, fonctionnelle et déontique d'organisations dans les SMA. In Mathieu, P. and Müller, J.-P., editors, *Systèmes multi-agents et systèmes complexes, JFIASMA'02*. Hermès.

IBM (2004). Hyper/J: Multi-dimensional separation of concerns for java. `http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm`.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, G., Loingtier, J., and Irwin, J. (1997). Aspect-oriented programming. In *1997 European Conference on Object Oriented Programming (ECOOP 97)*, pages 220–242. Springer.

Kniesel, G., Costanza, P., and Austermann, M. (2004). JMangler – a powerful back-end for aspect-oriented programming. In Filman, R., Elrad, T., Clarke, S., and Aksis, M., editors, *Aspect-Oriented Software Development*, chapter 9. Prentice Hall. to appear.

Lemaître, C. and Excelente, C. B. (1998). Multi-agent organization approach. In *Proceedings of the second Iberoamerican Workshop on Distributed Artificial Intelligence and Multi-Agent systems, Toledo, Spain*.

Lieberherr, K. J. (1996). *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston.

Müller, J.-P., Amiguet, M., Baez, J., and Nagy, A. (2001). La plate-forme MOCA: réification de la notion d'organisation au-dessus de madkit. In El Fallah Segrouchni, A. and Magnin, L., editors, *JFIADSMA'01*, pages 307–310.

Ossher, H. and Tarr, P. (2000). Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer.

Pinto, M., Fuentes, L., and Troya, J. M. (2002). Dynamic aspect-oriented platform. `http://www.lcc.uma.es/~pinto/AOP.htm`.

Popovici, A., Gross, T., and Alonso, G. (2002). Dynamic weaving for aspect oriented programming. In *1st International Conference on Aspect-Oriented Sortware Development (AOSD)*, Enschede.

Ricordel, P.-M. (2001). *Programmation Orientée Multi-Agents: Développement et Déploiement de Systèmes Multi-Agents Voyelles*. PhD thesis, Institut National Polytechnique de Grenoble.

Sassen, A.-M., Amorós, G., Donth, P., Geih, K., Jézéquel, J.-M., Odent, K., Plouzeau, N., and Weis, T. (2002). QCSS a methodology for the development of contract-aware components based on aspect oriented design. In *Early Aspects Workshop, AOSD 2002*, Enschede.

Schantz, R., Loyall, J., Atighetchi, M., and Pal, P. (2002). Packaging quality of service control behaviors for reuse. In *ISORC 2002, The 5th IEEE International Symposium on Object-Oriented Real-time distributed Computing.*

Schult, W. and Polze, A. (2003). Speed vs. memory usage - an approach to deal with contrary aspects. In *The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS).*

Sullivan, K., Gu, L., and Cai, Y. (2002). Non-modularity in aspect-oriented languages: integration as a crosscutting concern for AspectJ. In *The First International Conference on Aspect-oriented software development (AOSD)*, pages 19–26, Enschede, The Netherlands.

Sutton Jr., S. M. and Rouvellou, I. (2002). Modeling of software concerns in cosmos. In *1st International Conference on Aspect-oriented software development (AOSD)*, pages 127–133, Enschede, The Netherlands.

The AOSD Steering Committee (2004). aTrack project home. `http://www.aosd.net/`.

Wooldridge, M., Jennings, N. R., and Kinny, D. (2000). The Gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3:285–312.

Model checking of high-level object oriented specifications: the **L*f*P** experience[*]

Frédéric Gilliers†, François Bréant⋆, Denis Poitrenaud‡ & Fabrice Kordon‡

† SAGEM SA
Etablissement d'Eraggny, Avenue du Gros Chêne
95610 Eragny
B.P. 51 - 95612 Cergy Pontoise Cedex, France
frederic.gillier@sagem.com
⋆LaBRI, équipe vérification et test de systèmes informatiques
Université de Bordeaux 1
Domaine Universitaire
351, cours de la Libration
33405 Talence Cedex, FRANCE
francois.breant@lip6.fr
‡ Laboratoire d'Informatique de Paris 6/SRC
Université Pierre & Marie Curie
4, place Jussieu
F-75252 Paris CEDEX 05, France
denis.poitrenaud@lip6.fr, fabrice.kordon@lip6.fr

Abstract

The development of modern distributed applications is stretching the limits in term of design complexity and manageability. In order to develop reliable distributed applications new tools and methods must be developed. We propose a solution based on a modeling language **L*f*P** and and associated development methodology. This paper shortly presents our general methodology and the **L*f*P** language. It then focuses on formal verification of **L*f*P** specifications.

1 Introduction

The rapid advance of distributed technology has lead to systems stretching limits in terms of complexity and manageability [11]. This problem is crucial for reliable distributed systems which are required to have a deterministic behavior. To solve these development problems, it is of interest to consider development model based development approach [4].

Such an approach distinguishes two strong components [9]:

- a model on which any type of validation or verification techniques may be applied,
- the programs that implement this model and which are generated from it.

Such approaches become widely accepted under various names. As an example, MDA [13] (Model Driven Architecture) may be considered as a similar approach.

A distributed application is made of two orthogonal aspects: the control aspect, and the computational aspect. The control aspect manages the global state of the application whereas

[*]The work presented in this paper is being performed within the MORSE project. MORSE is a French government founded research project (RNTL) with industrial partners (Sagem, Aonix) and academic partners (LIP6 - Univ. P. & M. Curie, LaBRI - Univ. Bordeaux I).

the computational aspect covers the domain specific computational components. Model-Based development is of particular interest for distributed systems for two main reasons. First, they are very difficult to develop since they are very undeterministic ; thus, some apparently minor choices may have dramatic influences on the system behavior. Second, control aspects (the difficult part to build) strongly interact with both the execution environment and computational aspects, these interactions have to be carefully studied to avoid unexpected behaviors.

In that context, formal methods are of particular interest since they allow to prove the system using model checking techniques for example. However, [12] demonstrated that a major problem for the use of formal methods is the large education required by engineers to use them. The idea is then to encapsulate them using ”more common” languages for which no particular (or less) training is required. As an example, a similar approach is used in BLAST [6] that allows the use of C programs as inputs for model checking.

We have designed **L*f*P** (Language for Prototyping) [14], a formal notation dedicated to the specification, verification and code generation of distributed systems. **L*f*P** aims at providing a high level notation that fits the needs for describing the control part of a distributed system in a way that makes it usable for engineers. A first experience for verification from **L*f*P** specifications is described in [10] and [15]; it is based on a Petri net generated from the **L*f*P** specification for verification purposes.

This paper presents *direct model checking* on **L*f*P** specifications; which means that model checking is performed on the **L*f*P** specification instead of the corresponding Petri net like in [10, 15]. The symbolic model checker presented here is based on DDD (Data Decision Diagrams) that are an extension of BDD for discrete types [3]. We explain how we represent an **L*f*P** state using DDDs and the mechanisms we use to perform the state exploration. The key issues related to the formal verification of **L*f*P** specifications are related to the dynamic aspects of this language such as processes creation, RPC mechanisms, addressing, etc.

Section 2, briefly presents the overall methodology. We then present **L*f*P** in section 3 and finally detail in section 4 why DDD are well suited to represent **L*f*P** programs, our coding technique of **L*f*P** programs as well as issues and problems raised in this study.

2 Object Oriented Methodology

We propose a methodology to handle the specific issues of distributed applications. Its goal is to help the designer to achieve the development of a distributed application.

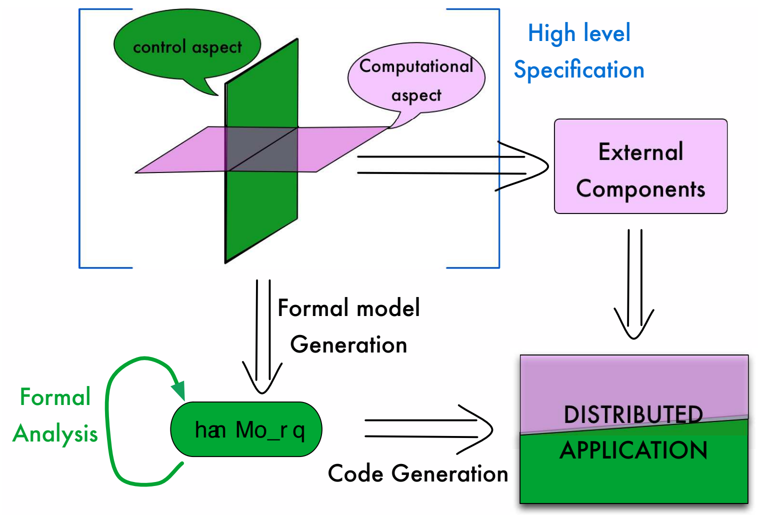


Figure 1: Development methodology associated to **L*f*P**

As shown on figure 1, our methodology relies on the separation between the control aspect and the data computation aspect in distributed systems. The control aspect handles the distribution of the application over the network, and the interaction protocols between the components of an application. The data computation aspect handles the domain specific calculus required to produce the results.

Our methodology starts with a high level description of the application written in UML. This specification should outline the interaction between the computational aspect and the control aspect of the system. We focus on the modelling of the control aspect since it is related to the specific issues identified in the development of distributed applications. Therefore, this control aspect of the specification is translated into **L*f*P** a formal language specifically designed to model distributed applications control aspect.

The interactions between the control aspect and the computational aspect are modeled using constructions of the **L*f*P** language very similar to private types defined in the Ada language [7].

The formal model obtained from the specification can then be formally checked against its requirements. State properties can be stated in the UML description in OCL, but properties which involve series of actions must be stated in temporal logic directly on the **L*f*P** specification. Formal verification of **L*f*P** specifications is the heart of this paper and will be fully described in section 4.

Once the model meets all its requirements, we provide a code generator for the **L*f*P** language. Automatic code generation translates the **L*f*P** semantics to provide an effective implementation of the specification and ensures the correctness of the implementation. It uses the description of the interactions between the control and the computational aspects to link the generated code to the external components. The underlying mechanism of code generation from an **L*f*P** model have been discussed in [5].

3 The L*f*P language

This section will present the **L*f*P** language through a simple client / server example. We show that **L*f*P** provides the appropriate abstractions to model interactions between components of an application.

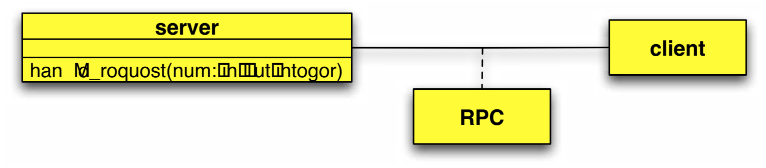


Figure 2: Simplified class diagram of the client server example

Let us first introduce our example with the simplified UML class diagram of figure 2 that shows the main model components. The client calls method `handle_request` on the server through a Remote Procedure Call implemented by the RPC interaction class. The server then returns a value through the `in out` parameter of this method. We will now present the **L*f*P** model corresponding to this system. First we will focus on the static description, then on the dynamic behavior of the components.

3.1 Static structure of the model

The static structure of an **L*f*P** model is described with an *architecture diagram*. Figure 3 shows the architecture diagram corresponding to the class diagram of figure 2. The main elements of the class diagram appear on the architecture diagram:

- interaction classes are translated into **L*f*P** *media* which are components that define the low level interaction protocol between the application components;
- classes of the model are translated into **L*f*P** *classes* which implements the control aspects in application components.

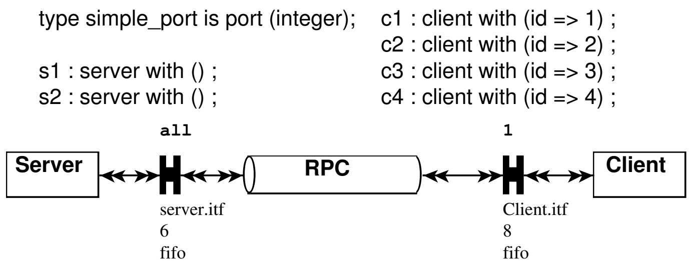

type simple_port is port (integer);
s1 : server with () ;
s2 : server with () ;
c1 : client with (id => 1) ;
c2 : client with (id => 2) ;
c3 : client with (id => 3) ;
c4 : client with (id => 4) ;
all
1
Server
RPC
Client
server.itf
6
fifo
Client.itf
8
fifo

Figure 3: Architecture diagram of the client / server example

In order to link the components and define the message queues of the application, the architecure diagram introduces *binders* to link the classes and media of the model. They formalize the message transmission between the components of the model. They are referenced in the components with variables of type `port` identified in the binder's binding attribute. These variables are of type `simple_port` defined in the diagram's definitions.

The architecture diagram also defines the static instances of the model: two instances of `server` and four instances of `client` are created on application start up. Each instance of the client has one of its attribute initialized with a value that identifies it.

On this specific architecture diagram, the binder that relates `RPC` to clients has multiplicity `1`, there is one binder instance for each instance of class `client`. The binder that relates `RPC` to `Server` has mutiplicity `all` which means that the binder is shared by all the instances of the class. A message in this binder may be read and handled by any server.

3.2 Dynamic behavior of model components

The dynamic behavior of the components is defined by their behavioral diagrams. This is an automaton that defines the actions performed by the **L*f*P** component in response to an event.

3.2.1 The `RPC` media

This media is in charge to implement the "Remote Procedure Call" protocol between the client and the server, it is shown on figure 4. This means that the media must send the message provided by the client to the server, read the return message from the server and send it to the client.

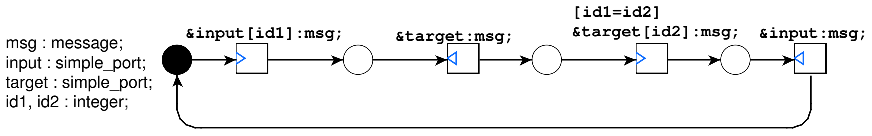


Figure 4: Behavioral diagram of media `RPC`

The media is related to client by `input` and to the server by `target`; both ports must be initialized by the component that creates the instance of `RPC`.

The first transition of the media reads a message on its `input` port and stores the discriminant in `id1`. This discriminant should identify the component that sent the message. Then the message puts the message in the `target` port.

The media then reads a message from the target port and only accepts it if its discriminant is equal to `id1`, that is if it was sent by the component that has sent the request. Finally the return message is sent back to the client, the media jumps to its initial state and is ready to handle a new message.

3.2.2 The `client` class

This class implements the client side of the system and is displayed on figure 5. This class first creates an instance of the `RPC` media to handle its communication with the server. Then it starts a loop that calls method `handle_request`. This means that a message requesting the execution of the method is put in the port `itf`, with a discriminant that contains the identifier of the client instance. The transition that contains this instruction is a `call` transition which also waits for the method's return message which updates the value of parameter `i`.

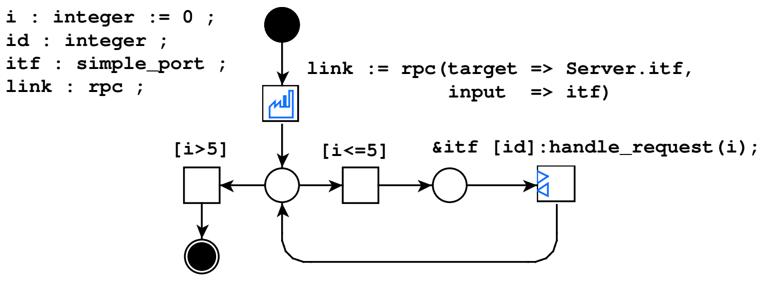
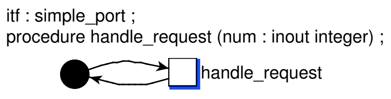


Figure 5: Behavioral diagram of class `client`

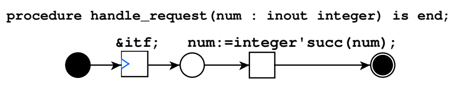
When the value of `i` becomes greater than five, the loops ends and the client finishes its execution.

3.2.3 The `server` class

The class `server` handles the request. Its main diagram is displayed on figure 6(a). When instanciated, a server keeps waiting for the activation of method `handle_request`. The behavior of this method is displayed on figure 6(b). It is activated by the arrival of an activation message on port `itf` which is the shared binder between the `RPC` media and the server class on the architecture diagram of figure 3.



(a) main diagram of the class



(b) behavioral diagram of method `handle_request`

Figure 6: Behavioral diagrams of class `server`

The method simply increases the value of its actual parameter and returns. Since the parameter mode is `inout`, the new value is sent through a return message to the caller. This message is implicitly sent on the method's activation port when the method returns. Since the discriminant of the return message is not specified, the discriminant of the activation message is used. In this case, it means that the discriminant of the return message contains the identifier of the client that sent the request.

4 Formal Verification

The verification of embedded distributed applications expressed in **L*f*P** covers a large number of properties. The verification process reduces to computing the reachability set of the program, which is the set of all possible states, and then evaluate assertions on the obtained set.

A data structure capable of representing large number of states must enable efficient operations such as equality test, set-theoretic operations, **L*f*P** specific operations, as well as a compact representation in memory.

We illustrate with a simple example a verification approach methodology based on the use of DDD (Data Decision Diagrams) for the symbolic computation of reachable states.

The first section introduces the DDD.

Section 2,3,4 successively describe the steps used for producing a verification program from an **L*f*P** specification:

- deriving an adequate model for verification purpose,
- computation of the reachable states,
- evaluation of assertions on the reachability set.

These steps are illustrated with the verification of the simple client/server application.

4.1 The Data Decision Diagrams (DDD)

The purpose of this paper is not to provide a complete definition of the DDD structure. Theoretical aspects of DDD are addressed in [3].

Data Decision Diagrams (DDDs) are *concise* data structures for representing *finite sets of assignment sequences* of the form $(e_1 := x_1; e_2 := x_2; \cdots; e_n := x_n)$ where e_i are variables and x_i are values. When an ordering on the variables is fixed and the variables are boolean, DDDs coincides with the well-know *Binary Decision Diagrams* [1, 2]. If an ordering on the variables is the only assumption, DDDs are the specialized version of the *Multi-valued Decision Diagrams* representing characteristic function of sets. For Data Decision Diagram, we assume no variable ordering and, even more, the same variable may occur many times in an assignment sequence, allowing the representation of dynamic structures: for a stack variable a, the sequence of assignments $(a := x_1; a := x_2; \cdots; a := x_n)$ may represent the stack content $x_1 x_2 \cdots x_n$.

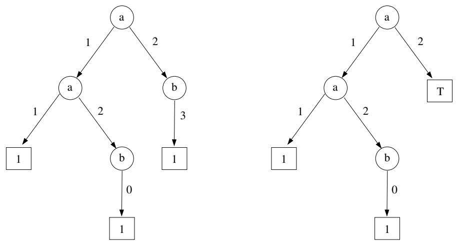


Figure 7: Two Data Decision Diagrams.

Traditionally, decision diagrams are often encoded as decision trees. Internal nodes are labeled with variables, arcs with values (of the adequate type) and leaves with either 0 or 1. Figure 7, left-hand side, shows the decision tree for the set $S = \{(a := 1; a := 1), (a := 1; a := 2; b := 0), (a := 2; b := 3)\}$ of assignment sequences. As usual, 1-leaves stand for accepting terminators and 0-leaves for non-accepting terminators. Since there is no assumption on the cardinality of the variable domains, we consider 0 as the default value. Therefore 0-leaves are not depicted in figure 7.

Unfortunately, any finite set of assignment sequences cannot be represented. Thus, we introduce a new kind of leaf label: $\top$ for *undefined*. Intuitively, $\top$ represents any finite set of assignment sequences. Figure 7, right-hand side, gives an approximation of the set $S \cup \{(a := 2; a := 3)\}$. Indeed, an ambiguity is introduced since after the assignment $a := 2$, two assignments have to be represented: $a := 3$ and $b := 3$. These two assignments affect two distinct variables so they can not be represented, as two distinct arcs outgoing from the same node cannot be labeled with the same value (in other words, non-determinism is not authorized in the decision tree).

However, since our verification approach only considers well formed assignments sequences with no compatibility issues regarding operations, we will consider $\top$ leafs as the result of an error in the state set computation. We now give an overview of DDDs, for a more formal and detailed presentation including theoretical aspects regarding $\top$, we refer the reader to [3].

4.1.1 Syntax and semantics of DDDs

In the following, E denotes a set of *variables*, and for any e in E, $\mathrm{Dom}(e)$ represents the *domain* of e.

Definition 1 (Data Decision Diagram) *The set* $\mathbb{D}$ *of DDDs is inductively defined by* $d \in \mathbb{D}$ *if:*

- $d \in \{0, 1, \top\}$ *or*
- $d = (e, \alpha)$ *with:*
  - $e \in E$
  - $\alpha : \mathrm{Dom}(e) \to \mathbb{D}$*, such that* $\{x \in \mathrm{Dom}(e) \,|\, \alpha(x) \neq 0\}$ *is finite.*

We denote $e \xrightarrow{z} d$*, the DDD* (e, α) *with* $\alpha(z) = d$ *and* $\alpha(x) = 0$ *for all* $x \neq z$.

Intuitively, a DDD can be seen as a tree. DDDs 0, 1 and $\top$ are leaves, and a DDD of the form (e, α) is a tree whose root is labeled with variable e, and with an outgoing arc labeled with x to a subtree $\alpha(x)$ foreach value $x \in \mathrm{Dom}(e)$. From a practical point of view, as non-accepting branches (i.e. branches ending with a 0-leaf) are not encoded, the "finite support" condition for α ensures that DDDs can be implemented (even when variables range over infinite domains).

The *meaning* $[\![d]\!]$ of a DDD d is a set of finite sets of assignment sequences. In particular, $[\![\top]\!]$ is the (infinite) set of all finite sets of asssignment sequences. When $\top$ does not appear in a DDD, the DDD represents a unique finite set of assignment sequences (i.e. its meaning is a singleton). Hence, such a DDD yields an exact (non approximate) representation and it is called *well-defined*.

The unique set in the meaning of a well-defined DDD d is the set of assignment sequences corresponding to accepting branches (i.e. branches ending with a 1-leaf) in the tree representation of d. In particular, we have $[\![0]\!] = \{\emptyset\}$ and $[\![1]\!] = \{\{()\}\}$ (where () is the empty sequence of assignments).

Equivalence checking for DDDs is crucial when DDDs are used to represent sets of states. Fortunately, DDDs admit *canonical forms* so that equivalence checking for DDDs in canonical form reduces to (syntactic) equality.

Intuitively, from the tree representation point of view, the canonical form of a DDD is obtained by replacing with 0 all sub-trees that have only 0-leaves and by sharing all subtrees which are equivalent. Two DDDs in canonical form are equivalent if and only if they are equal. Moreover, every DDD is equivalent to a DDD in canonical form.

In the following, *we only consider DDDs that are in canonical form.*

4.1.2 Operations on DDDs

First, we generalize the usual set-theoretic operations – *sum* (union), *product* (intersection) and *difference* – to finite sets of assignment sequences expressed in terms of DDDs. The crucial point of this generalization is that all DDDs are not well-defined and furthermore that the result of an operation on two well-defined DDDs is not necessarily well-defined. The *sum* $+$, the *product* $*$ and the *difference* $\setminus$ of two DDDs are inductively defined in the following tables. In these tables, for any $\diamond \in \{+, *, \setminus\}$, $\alpha_1 \diamond \alpha_2$ stands for the mapping in $\mathrm{Dom}(e_1) \to \mathbb{D}$ defined by $(\alpha_1 \diamond \alpha_2)(x) = \alpha_1(x) \diamond \alpha_2(x)$ for all $x \in \mathrm{Dom}(e_1)$.

$+$	0	1	$\top$	(e_2,α_2)
0	0	1	$\top$	(e_2,α_2)
1	1	1	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$
(e_1,α_1)	(e_1,α_1)	$\top$	$\top$	$(e_1,\alpha_1+\alpha_2)$ if $e_1=e_2$ $\top$ if $e_1\neq e_2$

$*$	0	1	$\top$	(e_2,α_2)
$0\vee(e_1,\alpha_1)\equiv 0$	0	0	0	0
1	0	1	$\top$	0
$\top$	0	$\top$	$\top$	$\top$
(e_1,α_1)	0	0	$\top$	$(e_1,\alpha_1*\alpha_2)$ if $e_1=e_2$ 0 if $e_1\neq e_2$

$\setminus$	0	1	$\top$	(e_2,α_2)
0	0	0	0	0
1	1	0	$\top$	1
$\top$	$\top$	$\top$	$\top$	$\top$
(e_1,α_1)	(e_1,α_1)	(e_1,α_1)	$\top$	$(e_1,\alpha_1\setminus\alpha_2)$ if $e_1=e_2$ (e_1,α_1) if $e_1\neq e_2$

These set-theoretic operations on DDDs actually produce the best possible approximation of the result. More precisely, if d and d' are two DDDs, then the sum $d+d'$ (resp. the product $d*d'$, the difference $d\setminus d'$) is the “best defined” DDD whose meaning contains the set $\{S\cup S' \mid S\in[\![d]\!] \text{ and } S'\in[\![d']\!]\}$ (resp. the set $\{S\cap S' \mid S\in[\![d]\!], S'\in[\![d']\!]\}$, the set $\{S\setminus S' \mid S\in[\![d]\!]$ and $S'\in[\![d']\!]\}$).

The concatenation operator defined below corresponds to the concatenation of language theory.

$$d\cdot d' = \begin{cases} 0 & \text{if } d=0\vee d'=0 \\ d' & \text{if } d=1 \\ \top & \text{if } d=\top\wedge d'\neq 0 \\ (e,\alpha\cdot d') & \text{if } d=(e,\alpha) \end{cases}$$

Notice that any DDD may be defined using constants 0, 1, $\top$, the elementary concatenation $e\xrightarrow{x}d$ and operator $+$, as shown in the following example.

Example 1 *Let d_A be the DDD represented in left-hand side of Fig. 7, and d_B the right-hand side one.*

$$\begin{aligned} d_A &= a\xrightarrow{1}\left(a\xrightarrow{1}1+a\xrightarrow{2}b\xrightarrow{0}1\right)+a\xrightarrow{2}b\xrightarrow{3}1 \\ d_B &= a\xrightarrow{1}\left(a\xrightarrow{1}1+a\xrightarrow{2}b\xrightarrow{0}1\right)+a\xrightarrow{2}\top \end{aligned}$$

Let us now detail some computations:

$$\begin{aligned} d_A+a\xrightarrow{2}a\xrightarrow{3}1 &= a\xrightarrow{1}\left(a\xrightarrow{1}1+a\xrightarrow{2}b\xrightarrow{0}1\right)+a\xrightarrow{2}\left(b\xrightarrow{3}1+a\xrightarrow{3}1\right) \\ &= a\xrightarrow{1}\left(a\xrightarrow{1}1+a\xrightarrow{2}b\xrightarrow{0}1\right)+a\xrightarrow{2}\top \\ &= d_B \\ \left(a\xrightarrow{1}1*a\xrightarrow{2}1\right)*\top &= 0*\top=0\neq a\xrightarrow{1}1*\left(a\xrightarrow{2}1*\top\right)=a\xrightarrow{1}1*\top=\top \\ d_A\setminus d_B &= a\xrightarrow{2}\left(b\xrightarrow{3}1\setminus\top\right)=a\xrightarrow{2}\top \\ d_B\cdot c\xrightarrow{4}1 &= a\xrightarrow{1}\left(a\xrightarrow{1}c\xrightarrow{4}1+a\xrightarrow{2}b\xrightarrow{0}c\xrightarrow{4}1\right)+a\xrightarrow{2}\top \end{aligned}$$

4.1.3 Homomorphisms on DDDs

In order to iteratively compute the reachability set of an **L*f*P** program, we need to translate **L*f*P** instructions into DDD operations. These complex operations on DDDs are described by homomorphisms. Basically, an homomorphism is any mapping $\Phi : \mathbb{D} \to \mathbb{D}$ such that $\Phi(0) = 0$ and such that $\Phi(d_1) + \Phi(d_2)$ is better defined than $\Phi(d_1 + d_2)$ for every $d_1, d_2 \in \mathbb{D}$. The sum and the composition of two homomorphisms are homomorphisms.

So far, we have at one's disposal the homomorphism $d * \mathrm{Id}$ which allows to select the sequences belonging to the given DDD d; on the other hand we may also remove these given sequences, thanks to the homomorphism $\mathrm{Id} \setminus d$. The two other interesting homomorphisms $\mathrm{Id} \cdot d$ and $d \cdot \mathrm{Id}$ permit to concatenate sequences on the left or on the right side. For instance, a widely used left concatenation consists in adding a variable assignment $e_1 = x_1$ that is denoted $e_1 \xrightarrow{x_1} \mathrm{Id}$. Of course, we may combine these homomorphisms using the sum and the composition.

However, the expressive power of this homomorphism family is limited; for instance we cannot express a mapping which modifies the assignment of a given variable. A first step to allow user-defined homomorphism Φ is to give the value of $\Phi(1)$ and of $\Phi(e \xrightarrow{x} d)$ for any $e \xrightarrow{x} d$. The key idea is to define $\Phi(e, \alpha)$ as $\sum_{x \in \mathrm{Dom}(e)} \Phi(e \xrightarrow{x} \alpha(x))$ and $\Phi(\top) = \top$. A sufficient condition for Φ being an homomorphism is that the mappings $\Phi(e,x)$ defined as $\Phi(e,x)(d) = \Phi(e \xrightarrow{x} d)$ are themselves homomorphisms. For instance, $inc(e,x) = e \xrightarrow{x+1} Id$ and $inc(1) = 1$ defines the homomorphism which increments the value of the first variable. A second step introduces induction in the description of the homomorphism. For instance, one may generalize the increment operation to the homomorphism $inc(e_1)$, which increments the value of the given variable e_1. A possible approach is to set $inc(e_1)(e,x) = e \xrightarrow{x+1} Id$ whenever $e = e_1$ and otherwise $inc(e_1)(e,x) = e \xrightarrow{x} inc(e_1)$. Indeed, if the first variable is e_1, then the homomorphism increments the values of the variable, otherwise the homomorphism is inductively applied to the next variables.

The formal definition of inductive homomorphisms can be found in [3]. The two following examples illustrate the usefulness of these homomorphisms to design new operators on DDD. The first example formalizes the increment operation. The second example is a swap operation between two variables. It gives a good idea of the techniques used to design homomorphisms for some variants of Petri net analysis.

Example 2 *This is the formal description of increment operation:*

$$\begin{array}{lcl} inc(e_1)(e,x) & = & \begin{cases} e \xrightarrow{x+1} Id & \text{if } e = e_1 \\ e \xrightarrow{x} inc(e_1) & \text{otherwise} \end{cases} \\ inc(e_1)(1) & = & 1 \end{array}$$

Let us now detail the application of inc over a simple DDD:

$$\begin{array}{lcl} inc(b)(a \xrightarrow{1} b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) & = & a \xrightarrow{1} inc(b)(b \xrightarrow{2} c \xrightarrow{3} d \xrightarrow{4} 1) \\ & = & a \xrightarrow{1} b \xrightarrow{3} Id(c \xrightarrow{3} d \xrightarrow{4} 1) \\ & = & a \xrightarrow{1} b \xrightarrow{3} c \xrightarrow{3} d \xrightarrow{4} 1 \end{array}$$

Example 3 *The homomorphism $swap(e_1,e_2)$ swaps the values of variables e_1 and e_2. It is designed using three other kinds of homomorphisms: $rename(e_1)$, $down(e_1,x_1)$, $up(e_1,x_1)$. The homomorphism $rename(e_1)$ renames the first variable into e_1; $down(e_1,x_1)$ sets the variable e_1 to x_1 and copies the old assignment of e_1 in the first position; $up(e_1,x_1)$ puts in the second position the assignment $e_1 = x_1$.*

$$
\begin{array}{lcl}
swap(e_1,e_2)(e,x) & = & \left\{ \begin{array}{ll} rename(e_1) \circ down(e_2,x) & if\, e=e_1 \\ rename(e_2) \circ down(e_1,x) & if\, e=e_2 \\ e \xrightarrow{x} swap(e_1,e_2) & otherwise \end{array} \right. \\
swap(e_1,e_2)(1) & = & \top \\
 & & \\
rename(e_1)(e,x) & = & e_1 \xrightarrow{x} Id \\
rename(e_1)(1) & = & \top \\
 & & \\
down(e_1,x_1)(e,x) & = & \left\{ \begin{array}{ll} e \xrightarrow{x} e \xrightarrow{x_1} Id & if\, e=e_1 \\ up(e,x) \circ down(e_1,x_1) & otherwise \end{array} \right. \\
down(e_1,x_1)(1) & = & \top \\
 & & \\
up(e_1,x_1)(e,x) & = & e \xrightarrow{x} e_1 \xrightarrow{x_1} Id \\
up(e_1,x_1)(1) & = & \top
\end{array}
$$

Let us now detail the application of swap over a simple DDD *which enlights the role of the inductive homomorphisms:*

$$
\begin{array}{lcl}
swap(b,d)(a\xrightarrow{1}b\xrightarrow{2}c\xrightarrow{3}d\xrightarrow{4}1) & = & a\xrightarrow{1}swap(b,d)(b\xrightarrow{2}c\xrightarrow{3}d\xrightarrow{4}1) \\
 & = & a\xrightarrow{1}rename(b)\circ down(d,2)(c\xrightarrow{3}d\xrightarrow{4}1) \\
 & = & a\xrightarrow{1}rename(b)\circ up(c,3)\circ down(d,2)(d\xrightarrow{4}1) \\
 & = & a\xrightarrow{1}rename(b)\circ up(c,3)(d\xrightarrow{4}d\xrightarrow{2}1) \\
 & = & a\xrightarrow{1}rename(b)(d\xrightarrow{4}c\xrightarrow{3}d\xrightarrow{2}1) \\
 & = & a\xrightarrow{1}b\xrightarrow{4}c\xrightarrow{3}d\xrightarrow{2}1
\end{array}
$$

One may remark that $swap(b,e)(a\xrightarrow{1}b\xrightarrow{2}c\xrightarrow{3}d\xrightarrow{4}1) = a\xrightarrow{1}\top$.

Basically, the DDD data structure provides a compact encoding and a usage of memory similar to the BDD (Binary Decision Diagram). The operations calls and their results are stored in an operation cache. Data are stored in a unicity table based on a hash table. A canonical representation of the states allows for efficient comparison of diagrams. Set-theoretic operations are defined and families of inductive homomorphisms representing the operations of the **L*f*P** language can be user defined. The DDD structure meets both the requirement of efficiency and compacity.

4.2 Deriving an adequate model for verification purpose

The direct computation of reachable states from the initial model may be impossible in many cases. The combinatory explosion can have different origins: inherent complexity of the problem, level of detail of the model, size of the model, concurrency.

One main difficulty is to evaluate the complexity of the model and identify modifications in order to enable reachability set computation. Different models may have to be derived in accordance to the properties to be verified.

We distinguish two ways of making changes to the model: quantitative and qualitative modifications. The only purpose of these modification is to focus on the search of potential problems.

4.2.1 Quantitative alteration

The quantitatives modifications often lead to a redimensionning of data. Such modifications are easy to achieve if the size of components are clearly visible in the design. The choice is guided by the effect of the change on the complexity and by considering the symmetries of the system. It may also permit to study different phases in the execution of the model, for instance initialisation, termination.

Example 4 *For instance, the simple model presented in this article depicts a client/server application composed of 4 clients communicating with 2 servers. Each client invokes 5 times a service that increment of 1 a parameter value. The service is supported by both servers, any available server can execute the service.*

If we call F(M, S, C) the number of possibles sequences of M messages by C clients to S server, a preliminary complexity analysis gives:

$F(M,1,C)$ $= F(M,1,(C-1)) * (((C-1)*M+1)^M)$
$> F(M,1,(C-1)) * (((C-1)*M)^M)$
$> F(M,1,(C-2)) * (((C-2)*M)^M) * (((C-1)*M)^M)$
$> (((C-1)! * M)^M)^{(c-1)}$

F(M, S, C) $> S^{F(M,1,C)}$

In the case of our example, we reach a number of sequences F(6, 2, 4)= 2^1.03 e+28 that doesn't even considers the additional complexity from the model. It becomes necessary to modify the model in order to compute the reachable states. We have been able to verify properties by reducing the size of the message sequence in various ways.

However, resizing the application for verification purpose may not be an acceptable solution. We currently study other approaches using abstract representations of set of states by means of DDD, [16].

4.2.2 Qualitative alteration

The qualitative changes include:

- simplification of the model that preserves information of the properties to be verified,
- addition of working hypothesis often linked to the domain of application and restricting the set of reachable states.

The simplification of the model often leads to the abstraction of some mechanisms. For instance, the merging of two transitions when conditions allow it, will reduce the interleaving and thus, will reduce the size of the reachability set. Such simplification may be acceptable when the validity of properties remain unaffected.

A working hypothesis defines what is a consistent state and what is a discardable state. Applying a working hypothesis on a set of states filters all inconsistent states with respect to the hypothesis.

Example 5 *For instance, additional hypothesis have been added to the initial model of a train system [8]. This model was describing the interactions and the behavior of a set of trains. The global state of the system was composed of the state of each train. The model was quite complex (>100 transitions) and its parallelism complicated the computation of the reachability set when dealing with a complex scenario. As time was not explicitly part of the model, valid states computed by firing sequences in the model would never exist in reality.*

The hypothesis that has been added had the effect of discarding all global states that consider the positions of trains at different dates. The application of this hypothesis had an immediate effect on state computation and allowed the verification of complex situations.

4.3 Computation of the reachable states

The computation of the reachable states consists in encoding the state of an **L*f*P** program by means of DDD and implementing the homomorphisms that compute the new states corresponding to the firing of transitions in the model.

4.3.1 Coding of a state

The state of an **L*f*P** specification is encoded into a DDD. Variables of the system are variables of the DDD and their values are attached to the arcs. The coding of the state represents

Type declaration of v	DDD
Array v;	$v \xrightarrow{elt1} v \xrightarrow{elt2} ...$
Set v;	$v \xrightarrow{size} v \xrightarrow{elt1} v \xrightarrow{elt2} ...$
VectorMultiSet v;	$v \xrightarrow{nbvect} v \xrightarrow{vect1size1} v \xrightarrow{vect1elt1} v \xrightarrow{vect1elt2} ...$

Table 2: Data types representation by means of DDD.

a mapping of variables that can only be accessed sequentially by the inductive homomorphisms. Thus, the implementation of homomorphisms strongly depends on the coding of the state. Three important issues have to be solved in the encoding of an **L*f*P** state by means of DDD as done in [16]. First, a canonical form must be defined and conserved during the state computation. Second, the particular encoding of an **L*f*P** state must support several dynamic aspects. And last, the encoding of the state must guarantee the compatibility with theoretic-set operators.

Canonicity is a fundamental requirement that allows reachable state computation. A canonical representation of the state allows to reduce the comparison between DDD to a pointer comparison (O(1)). Canonicity is achieved by imposing construction constraints on all structures. The operations that manipulate the state or any structure within the state produce states that respect the constraints. For instance, an absolute order between elements of a set must be defined. Operations on a sets must respect the order.

The coding of the state requires a coding of the data structures and associated operations supported by the input language that respect the canonical form. In the case of **L*f*P**, we need to implement the following features:

- multi-sets of vectors,
- sets and multi-sets of scalars,
- instances of classes and media,
- variables at all scopes (global, instance, local and class),
- arrays,
- stacks.

Dynamicity causes the handling of states with varying size while respecting the constraints of canonicity and operation compatibility. **L*f*P** provides dynamic features that are integrated in the state encoding:

- creation and destruction of instances,
- structures such as sets and multi-sets, FIFOs,
- method of function calls (stack).

By definition of the DDD structure, only one arc with a given value can be the output of a node. Thus, dynamicity can cause compatibility problems when adding DDD.

Example 6 *The following example recalls a typical case of incompatibility. The addition of the 2 DDD produces an invalid result where the node 'a' get 2 arcs valued '$V1$'.*
$a \xrightarrow{v1} a \xrightarrow{v3} c \xrightarrow{v4} ... + a \xrightarrow{v1} c \xrightarrow{v2} e \xrightarrow{v5} ... = a \xrightarrow{v1} \top$

We used prefixing techniques to construct DDD that are always compatible. For instance, a set is identified by one variable 'v'. A DDD representing the set 'v' can be seen as an assignment sequence using the same variable. Prefixing a set with a variable containing the size of the set will enforce the compatibility between DDD containing sets.

Typically, arrays and structures are coded using one variable. The semantic of a value (size, value, ...) depends on its position. The inductive homomorphisms use the position of the value to decode the structures.

The table 2 summarizes the encoding of some common structures.

Now that we can code all basic structures by means of DDD, we can show the encoding of the whole state. At the top of the hierarchy the different components of an **L*f*P** program appear in the following order:

1. global variables,
2. global binders (binder all),
3. instances,
4. end of state (special marker).

Each component is a DDD that represents a structure or a single variable like, for example, a special marker. The different fragments represented here are then assembled using the concatenation operator.

A canonical representation requires a particular ordering of object instances: the instances are grouped by class and an order must be defined in such a way that order of the instance remains the same whatever is the sequence of insertions. All object instances have the same structure shown below:

1. begin instance (special marker),
2. instance marker,
3. local media,
4. local binders,
5. instance variables,
6. program counter (*PC*),
7. stack (empty if not within a call),
8. end of instance marker.

The structure of the local media is simple:

1. media id (variable *me*),
2. message storage (multiset of vectors),
3. program counter (*PC*),

A block pushed on the stack is defined for each method. This block represents local data used by the method call. The corresponding DDD has the following shape:

1. parameters,
2. local variables,
3. return state.

Example 7 *The following DDD is an example of a global state of the client/server application with 1 client and 1 server. The variables prefixed with 'Glb' are the global variables, then comes the global binders and then the instances. In some cases, the use of symbols instead of values has been used to improve the legibility. Variables of the media in the client instance are easily recognized with the prefix 'RPC'.*

$$
\begin{array}{l}
Glb.tmp1 \xrightarrow{0} Glb.tmp2 \xrightarrow{0} Glb.tmp3 \xrightarrow{0} Glb.tmp4 \xrightarrow{0} \\
binder\_all\_in \xrightarrow{0} binder\_all\_out \xrightarrow{0} \\
\_\_BeginOfInstance \xrightarrow{server.mk} \\
server.mk \xrightarrow{0} server.iv.\_discr \xrightarrow{0} server.pc \xrightarrow{\_none\_begin} \\
\_\_EndOfInstance \xrightarrow{0} \\
\_\_BeginOfInstance \xrightarrow{CLIENT.mk} \\
CLIENT.mk \xrightarrow{0} \\
RPC.me \xrightarrow{0} RPC.loc\_set \xrightarrow{0} RPC.loc.discr \xrightarrow{1} RPC.loc.id1 \xrightarrow{0} RPC.loc.id2 \xrightarrow{0} \\
RPC.pc \xrightarrow{\_none\_begin} \\
CLIENT.locbinder \xrightarrow{0} CLIENT.locbinder\_out \xrightarrow{0} \\
CLIENT.iv.i \xrightarrow{0} CLIENT.iv.id \xrightarrow{1} \\
CLIENT.pc \xrightarrow{\_none\_begin} \\
\_\_EndOfInstance \xrightarrow{0} \\
\_\_EndOfState \xrightarrow{0} 1
\end{array}
$$

4.3.2 A set of basic Homomorphisms for L*f*P

A set of basic homomorphisms has been implemented to realize simple operations of the **L*f*P** language. The composition operation ($\circ$) allows to combine the operations in order to design the often complex homomorphism that represents the firing of a transition. A basic homomorphism takes as input a set of states and produces a set of intermediate states.

We summarize in the table 4 the homomorphisms that deal with the functionning of the state machines and the evaluation of boolean guards. Table 6 describes the homomorphisms that deal with communications and media.

Typically, an homomorphism associated to a transition starts by checking if all boolean conditions are satisfied, using the *MarkFireable* homomorphism. *MarkFireable* creates a new state where one instance that satisfies the condition is marked. If multiple instances qualify, then a state will be created for each one of them. All the other homomorphisms use the mark to identify the instance to be processed.

When the computation of the new states obtained by the firing of a transition is finished, the homomorphism *ResetMark* concludes the firing by reseting all markers from the new states.

State Machine Homomorphisms	***Parameters***	***Description***
MarkFireable	Class, ExprG, ExprL	Mark exactly one instance of an object of **Class** if its state satisfies the global boolean expression **ExprG** and the local boolean expression **ExprL**. Returns an empty set of states if no instance qualifies.
AssignVar	Class, Var, Expr	Assigns to the variable **Var** the result of the evaluation of the expression **Expr**. Variables in the expression and the assigned variable are assumed to be in the scope of the marked instance of class **Class**.
ResetMark	Class	Reset the marker of a marked instance of **Class**. This homomorphism concludes any transition firing.
Goto	Class, State	Assign the program counter of the marked instance of **Class** with the new automaton state **State**.
ResetGlbTmp		Initialize all temporaries to 0, in order to avoid duplicated states.
VarAdd	Class, Var, Inc, Modulo	Increments the variable **Var** in the scope of the marked instance of **Class** with **Inc** and **Modulo** parameters.
ProcessKill	Class	Destroy (remove from the state) the marked instance of **Class**.
InsertInstance	Class	Insert an instance of **Class** in the state.

Table 4: Homomorphisms related to state machines.

4.3.3 Coding of a transition

In most of the cases, each transition is associated to an homomorphism. For performance reason, an homomorphism is associated to states with multiple output branches and processes all transitions at the head of each branch.

A transition homomorphism can be seen as the composition of a precondition evaluation homomorphism and a postcondition homomorphism.

The following fragment of code shows an example of a C++ function returning the composition of the two homomorphisms that constitutes the firing of the transition. The operator $\circ$ is implemented with the C++ operator &. Operands of & appear in the inverse order of their application.

The parameter *gs* is an instance of the class descriptor of the application. It provides the capabilities to generate the DDD structure corresponding to the initial state of the system and ways to refer to variables within the DDD structure.

```
GHom FIRE_SERVER_handle_request_succ(GenAppState &gs)
{
return
```

Communication Homomorphism	***Parameters***	***Description***
SendCallMeth	Class, Binder, Target, Meth, NbParam, Param List	Generate a message containing a remote call of procedure **Meth** by an instance of **Class** to **Target** and store it in **Binder**. Parameters **Param List** are in the scope of the initiating instance. The binder can be either multi set, FIFO, local or global.
SendCallVMeth	same as SendCall-Meth, Var	Same as **SendCallMeth**, except that the method is the value of a variable **Var** in the scope of the emitting instance.
ProcessReturn	Class, Binder, NbParam, Param. List	Process a return message concluding a remote procedure call (RPC) initiated by **Class** after reception of the message in **Binder**. Assign values stored in the message to the list of variables passed in the parameter list **Param. List**.
ReceiveMethCall	Class, Binder, Locals, NextState, ReturnState	Process the reception of an RPC message emitted by an instance of **Class** and stored in **Binder**. Save the context on the local stack of the receiving instance, push local variables **Locals** and **ReturnState**, set program counter value to **NextState**.
Select	Class, Binder, Branch Descr.	Process a branch (state with multiple output arcs) with boolean and message guards in an instance of **Class** . Messages are read from **Binder**. A descriptor **Branch Descr** specifies the precondition and postcondition starting each branch.
Binder2Media	Binder	Transfer messages from a **binder** to a media. The message contains the id of the destination. **Binder** can be multi set or FIFO, local or global.
PurgeMedia		Empty the Media
Media2Binder	Binder	Transfer messages from a Media to a Binder. **Binder** can be multi set, FIFO, local or global.

Table 6: Homomorphisms related to Communications.

```
   POST_SERVER_handle_request_succ(gs)&
   TEST_SERVER_handle_request_succ(gs);
}
```

According to the **L*f*P** specification, the second transition of method *handle_request* increments the value of *num* and terminates the method call (cf fig. 6(b)). For implementation purpose, name are given to objects: the transition is called *SERVER_handle_request_succ* and the state precondition of the transition is called *SERVER_handle_request_start*.

The following code implements the precondition and postcondition homomorphisms for this transition.

The *TEST_SERVER_handle_request_succ* homomorphism creates states with one marked instance of class *Server* that satisfies the transition precondition.

```
GHom TEST_SERVER_handle_request_succ(GenAppState &gs)
{
return
  MarkFireable<GenCSServer>(
          Cst(1), // no condition on global variables, always true
          (Var(GenCSServer::PC) ==   // precondition is only testing the state
           Cst(AllStates::SERVER_handle_request_start)));
}
```

The *POST_SERVER_handle_request_succ* homomorphism realizes the action associated to the transition. The composition starts by incrementing the local variable num by the constant 1.

Then a return message from RPC is generated. This message carries a discriminant value *gs.Server._discr* that was set when the RPC was initiated. It contains the identifier of the initiator. The identifier of the message is *AllStates* :: *SERVER__none_ret_handle_request* specifies the type of the message. The message contains only one parameter : *num*.

Note that the construction of *gs* follows the same hierarchy as the original **L*f*P** program. For example *gs.Server.handle_request.num* refers to the local variable of method *handle_request*. The value of the C++ variable contains the identifier of the corresponding variable in the DDD.

The *Pop* homomorphism restores the context at the end of the method call. The *AssignVar* homomorphism resets the value of a local variable in order to limitate the generation of states. The *ResetMark* homomorphism concludes the firing of the transition by reseting the instance marker.

```
GHom POST_SERVER_handle_request_succ(GenAppState &gs)
{
    ResetMark<GenCSServer>()&  // reset the mark concluding
    AssignVar<GenCSServer>(gs.Server._discr, Cst(0))& // reset this local variable
    Pop(gs.Server)&   // exit from the method call
    scSendCallMeth<GenCSServer>(true, gs.BinderAllId_out,
                                gs, gs.Server._discr,
                                AllStates::SERVER__none__ret_handle_request,
                                1, gs.Server.handle_request.num)&
    Add<GenCSServer>(gs.Server.handle_request.num, 1, -1);
}
```

4.3.4 Reachable states computation

The simple computation of the reachable state space is an iterative process where all transitions are applied during one iteration.

```
Begin
DDD CURRENT
Set of transitions TSET
Create the initial state CURRENT

DDD ACC=NULL
Repeat
    DDD OLD_CURRENT=CURRENT
    For each transition T in TSET:
        CURRENT=CURRENT+T(CURRENT)
        BOOL GotNew=(ACC+CURRENT!=ACC)
        ACC=ACC+CURRENT
        CURRENT=CURRENT-OLDCURRENT
Until (!GotNew)
Return ACC
End
```

A preliminary filtering allows to eliminate transitions that have no chance to be fired. In our case this preliminary filtering is done by testing the simple condition on the program counter of each instance. A static structure generated by the verification program gives the list of potentially firable transitions for each value of the program counter of any instance. The list of potentially firable transition is updated on the fly when new states are computed. This improvement made a noticeable difference in performance.

A number of algorithms can be applied for the computation of the reachable states. Their study is beyond the scope of this paper.

4.4 Verification of properties

The verification process handles generic properties of distributed systems such as liveness, search for deadlocks, coverage, and specific properties tied to the system such as assertions on variables of the state. In any case, the verification of a property consists in expressing it by means of an homomorphism that will be applied to the set of reachable states.

4.4.1 Searching for deadlocks

If *T1,.. Tn* are the transitions of the system and *Hom_T1,.. Hom_Tn* the corresponding homomorphisms. Let *Hom_Sum* be the sum *Hom_Sum=Hom_T1 + Hom_T2 + .. Hom_Tn*. *Hom_Sum* will return the *null* homomorphism if it applies to a blocking state.

We reproduce the output generated by the verification program in the computation of blocking states of the system with 2 servers, 2 clients sending 2 messages. To simplify the reading, the DDD containing the state of the system is displayed by putting between parenthesis the value associated to a variable. Comments have been added to improve legibility. In the following, only one instance of each class is shown to save space.

```
[ TERMINALS =
// global variables:
 Glb.tmp1(0) Glb.tmp2(0) Glb.tmp3(0) Glb.tmp4(0)

//global binders:
 binder_all_in(0)  binder_all_out(0)

// instances
 __BeginOfInstance(server.mk)
// marker
 server.mk(0)
//instance variables & program counter
 server.iv._discr(0)  pc=server.pc(_none_begin)
 __EndOfInstance(0)

... // others instance of server

 __BeginOfInstance(CLIENT.mk)
 CLIENT.mk(0)

//media of client, instance variables
 RPC.me(0) RPC.loc_set(0) RPC.loc.discr(1) RPC.loc.id1(1) RPC.loc.id2(1)
 pc=RPC.pc(_none_begin)

// client local binders
 CLIENT.locbinder(0)  CLIENT.locbinder_out(0)

//client instance variables
 CLIENT.iv.i(2) CLIENT.iv.id(1)

 pc=CLIENT.pc(_none_end)
 __EndOfInstance(0)

... // others instance of client

__EndOfState(0)]
```

Only one state has been found with the following characteristics:

- no communication are pending
- all servers are ready to process a request
- all clients are terminated

So the only blocking state is a final state as it is defined in the model. This model has no invalid deadlocks.

4.4.2 Coverage test

A routine showing a compact view of the state space shows all possible values assigned to all variables composing the state. A look at the program counter of all instances shows that all the states have been explored. Additional states may be added to the model to deal with transitions sharing the same input and output state. This test can also be useful to:

- estimate or determine the size of the communication buffers (binders),
- define the domain of variables

In this example we only show the results for one instance of each class.

```
[ COVERAGE =

Glb.tmp1(0) Glb.tmp2(0) Glb.tmp3(0) Glb.tmp4(0)

binder_all_in(0 1 2 ) binder_all_out(0 1 2 )
__BeginOfInstance(536870922 )
server.mk(0 )
server.iv._discr(0 1 2 )
server.pc(_none_begin handle_request_begin handle_request_start)
__EndOfInstance(0 )

...

__BeginOfInstance(536870931 )
CLIENT.mk(0 )
RPC.me(0 ) RPC.loc.discr(0 1 ) RPC.loc.id1(0 1 ) RPC.loc.id2(0 1 )
RPC.pc(_none_begin _none_p1 _none_p2 _none_p3 )

CLIENT.locbinder(0 1 ) CLIENT.locbinder_out(0 1 )

CLIENT.iv.i(0 1 2 ) CLIENT.iv.id(1 )

CLIENT.pc(_none_begin _none_start _none_process _none__wait_handler_request _none_end ),
__EndOfInstance(0 )

...

__EndOfState(0 )
]
```

4.4.3 Assertions on variables

An assertion on a variable can be easily translated into an homomorphism that evaluates the boolean expression corresponding to the negation of the assertion. This homomorphism is applied on the set of reachable states and should return the empty set, if the original assertion is satisfied.

Example 8 *The computation of states of the system with 2 servers, 2 clients sending 2 messages has been performed.*

We now would like to show that the set of states that satisfy the negation of the expression $((Client.PC = client\_none\_end) \Rightarrow (client.i = 2))$ *is empty.*

The computation of such a set can be easily done by building the homomorphism that will check the existence of such states. We directly reuse the MarkFireable homomorphism that allows for checking boolean expressions on states.

```
GHom CheckAssertTerm(GenAppState &gs){
return
  MarkFireable<GenCSClient>(
          Cst(1), // no condition on global variables, always true

          (Var(GenCSClient::PC) == Cst(AllStates::CLIENT__none_end))&&
          (Var(gs.Client.i)!!=2)
          );
}
```

5 Conclusion and future work

In this paper, we have presented how we could handle the model checking of **L*f*P**, a high-level specification language. The main problem raised in this study resides in dynamic aspects proposed in **L*f*P**: creation of processes, RPC mechanisms, addressing, etc.

DDD were successfully used for the analysis of Petri nets [3] and were also experimented successfully for **L*f*P**. To achieve model checking, a basic set of homomorphisms has been defined and implemented to enable the computation of the reachability set of an **L*f*P** specification. An execution environment supporting basic debugging capabilities and the computation of the reachable set of an **L*f*P** model has been implemented. We have started developing a code generator that translates an XML file containing the **L*f*P** program and generates C++ files that will be linked to the execution environment.

The efficiency of the DDD operations and the compacity of the structure allowed the computation of large reachability sets [8] of complex **L*f*P** models. We summarized the main technical difficulties in the representation of a **L*f*P** program by means of DDD: canonical representation of the state, support for dynamicity, diversity and complexity of the mechanisms to implement.

We also address complexity issues that may be solved by modification of the model. The derivation of models for specific verification purposes is currently mostly supported by the user expertise. We believe that an abstract representation of the state may help solving these issues [16].

Compiler optimization techniques will be developed to optimize the construction of homomorphisms. A language for the specification of properties needs to be developed as well as associated translation tools to generate the homomorphisms.

The development of such tools will provide a wide access to the efficiency of the DDD structure and will free the user from the tedious manual implementation of homomorphisms.

References

[1] B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

[3] J. M. Couvreur, E. Encrenaz, E. Paviot-Adet, D. Poitrenaud, and P. A. Wacrenier. Data decision diagram for Petri nets analysis. In *Proc. of ICATPN 2002*, volume 2360 of *LNCS*, pages 101–120. Springer Verlag, june 2002.

[4] Frédéric Gilliers, Fabrice Kordon, and Dan Regep. Proposal for a Model Based Development of Distributed Embedded Systems. In *2002 Monterey Workshop : Radical Innovations of Software and Systems Engineering in the Future*. Springer Verlag, 2002.

[5] Frédéric Gilliers, Fabrice Kordon, and Jean-Pierre Velu. Generation of distributed programs in their target execution environment. In *Proceedings of the 15th International Workshop on Rapid System Prototyping*, pages 90–97. IEEE Computer Society, 2004.

[6] Thomas Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 58–70. ACM, 2002.

[7] ISO. *Information Technology – Programming Languages – Ada*. ISO, February 1995. ISO/IEC/ANSI 8652:1995.

[8] F. Kordon and M. Lemoine, editors. *Formal Methods for Embedded Distributed Systems: How to Master the Complexity*. Kluwer Academic, 2004. ISBN:1-4020-7997-4.

[9] F. Kordon and Luqi. An Introduction to Rapid System Prototyping. *IEEE Trans. Softw. Eng.*, 28(9):817–821, 2002.

[10] Fabrice Kordon, Isabelle Mounier, Emmanuel Paviot-Adet, and Dan Regep. Formal verification of embedded distributed systems in a prototyping approach. In *Monterey Workshop 2001: on Engineering Automation for Software Intensive System Integration*, June 2001.

[11] N. Leveson. Software engineering: Stretching the limits of complexity. *Communications of the ACM*, 40(2):129–131, 1997.

[12] Luqi and J. Goguen. Formal methods: Promises and problems. *IEEE Software*, 14(1):73–85, January / February 1997.

[13] OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. Technical report, OMG, 2001.

[14] Dan Regep and Fabrice Kordon. LfP: A Specification Language for Rapid Prototyping of Concurrent Systems. In *Proceedings of the 12th International Workshop on Rapid System Prototyping*, pages 90–97. IEEE Computer Society, 2001.

[15] Dan Regep, Yann Thierry-Mieg, and Fabrice Kordon. Modélisation et vérification de systèmes répartis: une approche intégrée avec LfP. In *Proceedings of AFADL'03*, January 2003.

[16] Yann Thierry-Mieg, Jean-Michel Ilié, and Denis Poitrenaud. A symbolic symbolic state space representation. In *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE'04)*, Madrid, Spain, September 2004. Springer Verlag. To appear.

DIMA-Maude: Toward a Formal Framework for Specifying and Validating DIMA Agents

Farid Mokhati (1), **Noura Boudiaf** (1), **Mourad Badri** (2) **& Linda Badri** (2)

(1) Département d'Informatique
Université d'Oum El-Bouagui
Oum El-Bouagui, Algérie
Mokhati@yahoo.fr
Boudiafn@yahoo.com

(2) Département de Mathématiques et
et d'Informatique
Université du Québec à Trois-Rivières
Mourad_Badri@uqtr.ca
Linda_Badri@uqtr.ca

Abstract

We present, in this paper, a formal and generic framework, called DIMA-Maude, supporting a formal description and validation of DIMA (Development and Implementation of Multi-Agents systems) agents. The framework is described using the formal and object-oriented language Maude. Based on rewriting logic, the Maude language offers an interesting way for concurrent systems formal specification and programming. In its present version, the proposed DIMA-Maude framework allows describing DIMA communicating agents. Their meta-behavior is described using an ATN (Augmented Transition Network). The main objectives of our work are essentially: (1) to specify formally multi-agents systems' behavior, (2) to provide a sound description preserving the consistency in their behavior and (3) to develop an approach supporting their validation process. The Maude descriptions, generated in the framework of our approach, have been validated using the platform supporting the Maude language.

Key words: Multi-Agent Systems, Quality Assurance, DIMA, Communicating Agents, Maude, Formal Specification, Behavior, Validation.

1 Introduction

A large number of models, languages and architectures supporting the development process of Multi-Agents Systems (MAS) have emerged during the last decade. They allowed bringing interesting solutions to several problems related to the development and the validation of MAS. However, only few papers addressed their formal specification. In the field of agents' behavior specification, three major approaches emerge in the literature: state-charts based approaches [Tra99, Pau03], Petri Nets (PN) based approaches [Cos99, Bak00] and, finally, approaches representing adaptation of object-oriented specification methods [Ode00, Ode01]. Using such approaches, MAS' behavior is generally specified in a diagrammatic or semi-formal way. This weakness, combined with the inherent problems of MAS' specification, often generates several difficulties and problems in their development process. MAS' behavior needs to be clearly specified, validated and correctly implemented [Pau02]. The numerous underlying concepts of multi-agents models, as well as their complexity, require a powerful formal model in terms of description. Furthermore, this model must be able to support the description of diverse important aspects related to MAS' behavior, such as interactions between agents and their concurrence at different levels. The presence of a tool supporting the model, particularly for simulation and verification objectives, is in our opinion essential.

DIMA model's architecture [Gue03] is one of the interesting proposals in the field of MAS development. Thanks to its modularity features, DIMA helps to decompose the complex behavior of an agent within a set of specialized behaviors. Those diverse behaviors can eventually be described by a meta-behavior [Gue03]. Otherwise, DIMA model offers a rich library composed of basic bricks allowing the construction of various agents' models. Moreover, DIMA offers different frameworks (ATN-based framework, Rule-based framework, Case-based framework, etc). The main objective of our work is to develop a formal framework, using the Maude language, supporting the construction of formal models for DIMA agents. Our first interest is related to the communicating agents of the DIMA model. Their meta-behavior is described by an ATN (Augmented Transition Network).

Knowing the richness and the complexity of the DIMA model, we retained in our approach the formal and object-oriented language Maude [Mes92, Man99]. The constructs offered by this language allow capturing the multiple aspects of the DIMA model. Maude is in fact a formal language based on a sound and complete logic called the rewriting logic [Mes92]. On certain aspects, it is close to the language Troll [Saa93], also having a sound semantics. However, the absence of concurrence in Troll makes Maude more advantages. The G-ECATNets [Cha01], also represent another interesting and very expressive candidate. They present an interesting integration of ECATNets [Bet92] and of the object paradigm. The ECATNets are also high-level Petri Nets. They integrate abstract algebraic types and Petri Nets' formalism. The G-ECATNets have, in fact, a representation in the Maude language, making them a particular case of Maude. Furthermore, the G-ECATNets do not dispose of a tool independent of Maude. An execution of a description in G-ECATNets is exactly like executing their representation in Maude. In this paper, we demonstrate the feasibility as well as the interest of formalizing the behavior (individual and social) of the ATN-based communicating agents of the DIMA model using the Maude language. The elaborated formal approach allows capturing the inherent aspects of the DIMA model. The Maude descriptions generated in the framework of our approach have been validated using the platform supporting this language.

The remainder of the paper is organized as follows: Section 2 outlines the major related work. We briefly present, in section 3, the DIMA multi-agents model. In Section 4, we introduce briefly the rewriting logic as well as the Maude language. Section 5 illustrates the process we propose to formalize the DIMA communicating agents using Maude. Section 6 presents the application of our approach on a concrete case study. In section 7, we discuss our present work and give some conclusions and future work directions.

2 Related Work

We already mentioned, in the previous section, some approaches that have been proposed in the literature to describe multi-agents systems' behavior. The formalism supporting the RCA roles protocol's descriptions proposed in [Tra99] is also used to describe agents' behavior. It is based on statecharts including seven states and two transitions. These seven states are: the initial state, the final state, the elementary action state, the composite action state, the state of communication, and states of waiting. The two types of transitions are the internal transition and the external transition. It is easy, using this formalism, to recognize coordination points between dual protocols.

Cost and al. [Cos99] proposed an approach that allows representing the conversations between agents using Colored Petri Nets. They have explored the use of model-based conversation specification in the context of MAS supporting manufacturing integration [Pen99]. In such an approach, agents are developed using the Jackal agent development platform [Sco98]. They communicate using the KQML agent communication language [Tim97]. Moreover, one of the most recent approaches for modeling agent-specific features is the AUML [Ode00]. As part of AUML, the authors suggest an extension of the UML by introducing a new type of diagrams called *protocol diagrams*. These diagrams combine elements of UML interaction diagrams and state diagrams to model the roles that can be played by an agent while interacting with other agents. The introduced diagram allows specifying multiple threads within an interaction protocol and supporting protocol nesting and protocol templates based on generic protocol description. These different approaches represent certainly several elements of response to some problems related to MAS development, but it remains that they only offer a partial formalization of MAS.

Our approach is similar, in terms of objectives, to previously quoted approaches. Essentially, it consists of supporting the important step of the specification of agent's behavior. However, we preferred to adopt a more formal way in our approach because of the different advantages it procures. It is based on the formal and object-oriented Maude language [Mes92, Man99, McC03].

3 DIMA Multi-Agents Model

MAS' design, as mentioned by many authors, is still anticipated. One of the major reasons is related to the fact that no standard exists nowadays for an agent's definition or its internal architecture [Fra00]. Throughout our approach, we opted for the DIMA agents' model, proposed by Guessoum in [Gue03]. This model is illustrated in figure 1.

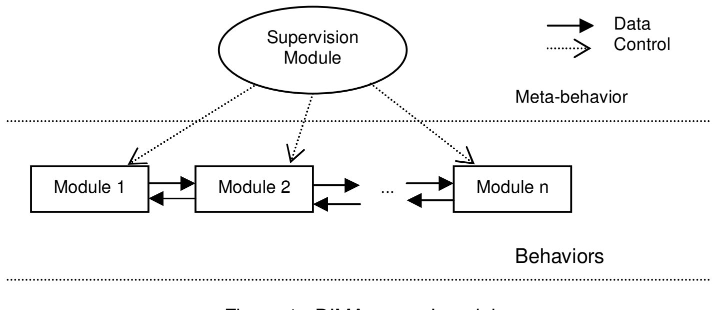


Figure 1 : DIMA agents' model

The DIMA model is a Java multi-agents platform built as an extension of Object-Oriented Programming. It provides a generic and modular agent architecture with several facilities (message passing, distribution, etc.), and allows high heterogeneity in agent architectures (reactive, deliberative and hybrid), and various customizations. The model of DIMA agents' architecture proposes to decompose an agent in different components whose goal is to integrate some existing paradigms, notably of artificial intelligence paradigm. An agent can have one or several components that can be reactive or cognitive.

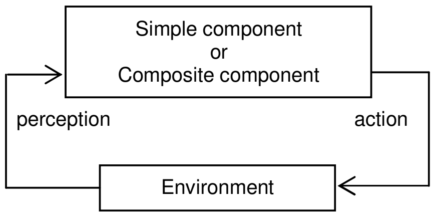


Figure 2: DIMA agents' architecture

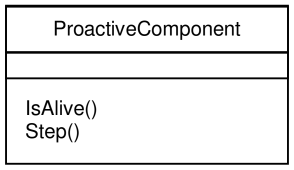


Figure 3: General structure of a proactive component

An agent is a simple component, or a composite component as illustrated by figure 2, that manages the agent's interaction with its environment and represents its internal behavior. The environment regroups all other agents as well as entities that are not agents. The basic brick of this architecture is represented by a proactive component (figure 3) on which are founded the different models of agents. A proactive component must describe:

- Goal: implicitly or explicitly described by the method *IsAlive ()*.
- Basic behaviors: a behavior could be implemented as a method.
- Meta-behavior: it defines how the behaviors are selected, sequenced and activated. It relies on the goal (use of the method *Step ()*).

3.1 ATN-Based Communicating Agents

A communicating agent is a proactive agent that introduces new functionalities of communication. Every communicating agent must have:

- A Mailbox to stock its messages.
- A communication component (figure 4) for managing the sending and the reception of messages.
- Possibly, a list of its acquaintances.

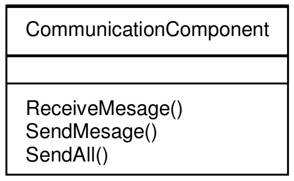


Figure 4: General Structure of a communication component

The meta-behavior of communicating agents is, in the adopted approach, described by an ATN. This last is defined by a finished set of states and transitions. A state of an ATN describes the agent's state [Gue96]. To define an ATN, we must:

- Define states: There are three types of states, initial (one), final (zero or more) and intermediate (zero or more).
- Define transitions: Every transition has conditions (one or more), actions (one or more) and a target state.
- Attach transitions to states.

The development of a proactive agent requires the description of its behaviors and its meta-behavior by inheriting the *ProactiveComponent* class. We must also describe the agent's goal by describing the method *IsAlive()*. In the case of agents whose meta-behavior is described by an ATN, such as ATN-based communicating agents, the method *IsAlive()* tests if the final state is not reached. For the same type of agents, the method *Step()*, describing a cycle of the meta-behavior of the proactive component, allows activating from the current state of an agent a transition whose condition is verified.

4 Rewriting Logic and Maude Language

4.1 Rewriting Logic

The rewriting logic, having a sound and complete semantics, was introduced by Meseguer [Mes92]. It allows describing concurrent systems. This logic unifies all the formal models that express concurrence [Mes90]. In rewriting logic, the logic formulas are called rewriting rules. They have the following form: *R:[t] → [t'] if C*. Rule *R* indicates that term *t* becomes (is transformed into) *t'* if a certain condition *C* if verified. Term *t* represents a partial state of a global state *S* of the described system. The modification of the global state *S* of the system to another state *S'* is realized by the parallel rewriting of one or more terms that express the partial states. The distributed state of a concurrent system is represented as a term whose sub-terms represent the different components of the concurrent state. The concurrent state's structure can have a variety of equivalent representations because it satisfies certain structural laws (equivalence class). For example, in an object-oriented system, the concurrent state that is usually called configuration has the structure of a multi-set of objects and messages. Therefore, we can see the constructed configurations by a binary operator applied to binary sets:

```
1. sort Configuration .
2. sort Object .
3. sort Msg .
4. subsort Object < Configuration .
5. subsort Msg < Configuration .
6. op null : -> Configuration .
7. op_ _ : Configuration Configuration -> Configuration [assoc comm id : null] .
```

Figure 5 : *Example of a portion* of the Maude program

The illustrated portion of program in figure 5 gives a definition of three types: *Configuration*, *Object* and *Msg*. In lines 4 and 5, *Object* and *Msp* are sub types of *Configuration*. Objects and messages are in fact multi-set configuration singletons. More complex configurations are generated from the application of the union on these multi-set singletons (objects and messages). Where there is neither floating messages nor live objects, we have in this case an empty configuration (line 6). The construction of a new configuration in terms of other configurations is done with line 7's operation. We can see that this operation has no name and that the two sub lines indicate the positions of two parameters of configuration type. This operation, which is the multi-set union, satisfies the structural laws of association and of commutation. It also possesses a neutral element *null*. For example, if we have a message *M1* that represents a configuration, and an object <*O : C*|*atts* > (please note that *O* an object's identifier, *C* the class to which it belongs and *atts* is the list of its attributes) that represents in itself another configuration, then we can construct another configuration in terms of those two configurations: *M1* < *O* : *C* | *atts* >. This one is equivalent to the configuration < *O* : *C* | *atts* > *M1* because the __ operation is commutative.

4.2 Maude

Maude is a specification and programming language based on the rewriting logic [Mes92, Man99]. Three types of modules are defined in Maude. The *functional* modules allow defining data types and their functions through equations theory. The *system* modules define the dynamic behavior of a system. This type of module extends the functional modules by the introduction of rewriting rules. A maximal degree of concurrence is offered by this type of module. Finally, there are the *object-oriented* modules that can be reduced to system modules. In relation to the system modules, the object-oriented modules offer a more appropriate syntax to describe the basic entities of the object paradigm, like messages or configuration for example. Only one rewriting rule allows expressing the consumption of certain floating messages, the sending of new messages, the destruction of objects, the creation of new objects, state change of certain objects, etc.

4.2.1 Functional modules

Functional modules define data types and their related operations using equation theories [Man03]. Functional modules support multiple sorts, subsort relations, operator overloading, and assertions of membership in a sort. A functional module is declared in Maude as follows:

fmod <*ModuleName*> is <*DeclarationsAndStatements*> endfm

4.2.2 System modules

A system module in Maude specifies a *rewrite theory*. A rewrite theory has sorts, kinds, and operators, and can have three types of statements : equations, memberships, and rules. A system module is declared in Maude as follows:

mod <*ModuleName*> is <*DeclarationsAndStatements*> endm

4.2.3 Object-oriented modules

Object-oriented modules define the concurrent object-oriented systems using a syntax more convenient than that a system module because it assumes acquaintance with the basic entities, such as objects, messages and configuration, and supports linguistic distinctions appropriate for the object-oriented case. An object-oriented module is declared in Maude as follows:

omod <*ModuleName*> is <*DeclarationsAndStatements*> endom

5 Modeling of Communicating Agents of the DIMA Model

We developed a formal framework allowing the construction of communicating agents. Their meta-behavior is described by an ATN. The framework is composed, as illustrated by figure 6, of several modules: an object-oriented module (*ATN-BASED COMMUNICATING-AGENT*) and several functional modules (the remainder of modules).

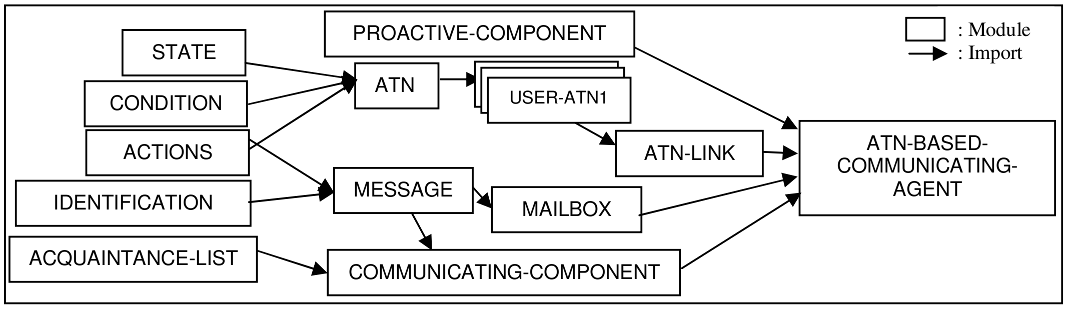


Figure 6 : DIMA-Maude frameworks' architecture.

- The functional module *STATE* (figure 7) contains the different necessary type declarations for the definition of a state (line [1]) and, on the other hand, the definition of operations used for the construction and the manipulation of a state (lines [2, 3, 4, 5, 6]), as well as equations implementing these operations (lines [7, 8, 9]).

```
(fmod STATE is
sorts State KindState NameState .                                   ***[1]

ops initial final ordinary : -> KindState .                         ***[2]
op AgentState : NameState KindState -> State .                      ***[3]
op IsInitial : State -> Bool .                                      ***[4]
op IsOrdinary : State -> Bool .                                     ***[5]
op IsFinal : State -> Bool .                                        ***[6]

var k : KindState .
var ns : NameState .

eq  IsInitial(AgentState(ns, k)) = if  k == initial then true        ***[7]
                else false fi .
eq  IsOrdinary(AgentState(ns, k)) = if  k == ordinary then true      ***[8]
                else false fi .
eq IsFinal(AgentState(ns, k)) = if  k == final then true             ***[9]
                else false fi .
endfm)
```

Figure 7: The functional module *STATE*.

- In module *CONDITIONS* (figure 8), in addition of the type *Condition*, we define the constant *NoCondition* of type Condition to indicate that a transition is unconditional.

```
(fmod CONDITIONS is
 sorts Condition .
 op NoCondition : -> Condition .
endfm)
```

Figure 8: The functional module *CONDITIONS*.

```
(fmod ACTIONS is
 sort  Action .
 op NoAction : -> Action .
 op IsInternalAction : Action -> Bool .
 op IsSendingAction : Action -> Bool .
 op IsReceivingAction : Action -> Bool .
 op IsSendingActionToAll : Action -> Bool .
endfm)
```

Figure 9: The functional module *ACTIONS*.

- In module *ACTIONS* (figure 9), in addition of the type Action, we define the constant *NoAction* that expresses the fact that at the time of the appearance of an event no action will be done. The *IsInternalAction* function serves to determine if an action is internal. The two *IsSendingActions* and *IsReceivingActions* operations are used to indicate if an action is of type communication (of sending reception) or no. Otherwise, the *IsSendingActionToAll* function indicates if it is necessary to send an action to all acquaintances of an agent.

- To define an ATN, we propose the *ATN* module (figure 10). This module reuses the *STATE*, *CONDITIONS* and *ACTIONS* modules. It includes the definition of two operations: *TargetState* that determines the state destination according to a state source and a condition, and the *AccomplishedAction* operation to determine the action executed according to a state and a condition.

```
(fmod ATN is
protecting STATE .
protecting CONDITIONS .
protecting ACTIONS
op TargetState : State Condition -> State .
op AccomplishedAction : State Condition -> Action .
endfm)
```

Figure 10: The functional module *ATN*.

- For the construction of an ATN for an application, we propose to spread the *ATN* module in another module *USER-ATN* (figure 11). In this module the user must:
  - mention all states constituting the ATN,
  - define all conditions and actions,
  - attach conditions and actions to states, using the two *TargetStates* and *AccomplishedActions* functions,
  - specify if action is type communication or no, using the *IsInternalActions*, *IsSendingActions*,

IsReceivingActions and *IsSendingActionToAll* functions are defined in module *ACTIONS*. To each category of agents (play the same role) is associated a module *USER-ATN*.

```
(fmod USER-ATN is
extending ATN .
***User part***
endfm)
```

Figure 11: The functional module *USER-ATN*.

- To respect the protocol of interaction used between agents, we propose to make a sort of link between the ATNs of different agents. This link consists in guaranteeing that at the time of the reception of a message, an agent cannot consume such a message except if it is in the corresponding state to the state of the agent sender. What implies is that there is a correspondence between different agent states. As a sending action accomplished by an agent sender represents an event for an agent receiver. Therefore there is a correspondence between sending actions of the sender and events received by the receiver. For that, the user must develop the ATN-LINK module (figure 12) that contains the correspondence on the one hand between different agent states, and on the other hand between actions of senders and conditions of receivers.

```
(fmod ATN-LINK is
protecting USER-ATN .
op  CorrspondingAgentState : State -> State .
op  CorrspondingCondition : Action ->Condition .

***User part***
endfm)
```

Figure 12: The functional module *ATN-LINK*.

```
(fmod IDENTIFICATION is
 sorts AgentIdentifier  .
 subsort  AgentIdentifier < Oid .
 endfm)
```

Figure 13: The functional module *IDENTIFICATION*.

- To describe the identification mechanism of agents, we define the functional module *IDENTIFICATION* (figure 13).

- The form adopted in our approach for messages is defined in the functional module *MESSAGE* (figure 14). This last imports modules *IDENTIFICATION* and *ACTIONS*.

```
(fmod MESSAGE is
 protecting IDENTIFICATION .
 protecting ACTIONS .
 sorts Message  Content .
 subsort Action < Content .
 op _:_:_ : AgentIdentifier  Content  AgentIdentifier -> Message .
endfm)
```

Figure 14: The functional module *MESSAGE*.

- The communicating agents are generally endowed with a Mailbox containing the received messages of other agents. For that, we define a functional module *MAILBOX* to manage Mailboxes of agents. This module imports the functional module MESSAGE.

- An agent must be endowed with a list of its acquaintances. In this way, it can exchange messages with other agents. For that, we define a functional module *ACQUAINTANCE-LIST* to manage lists of agent acquaintances. Due to limitation of space and the important size of these last two modules, we don't present them in this paper.

- The communicating agents are endowed with a module of communication that manages the exchange of messages between agents. In our approach, we define a functional module *COMMUNICATION-COMPONENT* (figure 15) that imports the two *ACQUAINTANCE-LISTS* and *MESSAGE* modules. In this module, we define three messages: *SendMessages* and *ReceiveMessages* that have as parameter a message of the form defined in the module *MESSAGE*.

```
(fmod COMMUNICATION-COMPONENT is
 protecting ACQUAINTANCE-LIST  .
 protecting MESSAGE .
 subsort acquaintance < AgentIdentifier .
 op SendMessage : Message -> Msg .
 op ReceiveMessage : Message -> Msg .
endfm)
```

Figure 15: The functional module *COMMUNICATION-COMPONENT*.

- Indeed, a communicating agent is first a proactive agent that has a goal to achieve described by the function *IsAlive* (figure 16). The *Parameters* and *Void* types are generic. They can be replaced by other types in modules wanting to reuse the *PROACTIVE-COMPONENT* module, provided that these new types are under-types of *Parameters* and/or *Void*. The basic cycle of the meta-behavior of proactive component described by method *step()* is modeled by rewriting rules of the module *ATN-BASED-COMMUNICATING-AGENT* of the figure 17.

```
(fmod PROACTIVE-COMPONENT is
 sort Parameters  Void .
 op IsAlive : Parameters -> Bool .
 op Step :   Parameters -> Void .
 endfm)
```

Figure 16: The functional module *PROACTIVE-COMPONENT*.

- The object-oriented module *ATN-BASED-COMMUNICATING-AGENT* (figure 17) is the main module. It imports the *PROACTIVECOMPONENTS*, *ATN-LINK*, *COMMUNICATION-COMPONENT* and *MAILBOX* modules. For a formal description of communicating agents, we propose the class Agent (line [1]). The definition of this class has the *CurrentStates*, *MBox*, and *AccList* attributes, to contain, in this order, the current state of the agent, its Mailbox and the list of its acquaintances. For ATN-based communicating agents, the description of the goal requires redefining the *IsAlive* function of the figure 16. We use a state as parameter of this function (line [5]) that returns a *Boolean*. This function returns the *Boolean* value true if the current state is final otherwise it returns false. The *State* type is an under-type of *Parameters*. The apparition of an event is expressed by message *Event* (line [2]) having as parameters an agent, a state and a condition. The execution of the corresponding action to this event is described either by the message *Execute* (line [3]) if it is internal, either by one of the two *SendMessages* or *ReceiveMessages* defined in the module *COMMUNICATION-COMPONENT*. The *GetEvent* message (line [4]) is a message used temporarily during the execution of rewriting rules.

```
(omod ATN-BASED-COMMUNICATING-AGENT is
protecting PROACTIVE-COMPONENT .
protecting ATN-LINK.
protecting COMMUNICATION-COMPONENT .
protecting MAILBOX .
subsort State < Paramaters .

class  Agent | CurrentState : State, MBox : MailBox, AccList : acquaintanceList  .       ***[1]
Msg Event :  AgentIdentifier State Condition -> Msg .                                     ***[2]
Msg Execute : Action -> Msg .                                                             ***[3]
Msg GetEvent : AgentIdentifier  State Condition -> Msg .                                  ***[4]
*******************************************************************************************************************
vars A  A1: AgentIdentifier . var NS : NameState . var KS : KindState . vars S S1 : State .
var Cond : Condition . var Act : Action . vars MB MB1 : MailBox . vars ACL ACL1   :  acquaintanceList .

eq IsAlive (AgentState(NS, KS)) = IsFinal(AgentState(NS, KS)) .                             ***[5]
*******************************************First case***************************************************************
crl [StepCaser1] :  Event(A, S, Cond) < A : Agent | CurrentState : S, MBox : MB, AccList : ACL >
                    => Execute(AccomplishedAction(S, Cond)) GetEvent(A, S, Cond)
                       < A : Agent | CurrentState : S, MBox : MB, AccList : ACL >
                       if (IsAlive(S) == false) and
                        (IsInternalAction(AccomplishedAction(S, Cond)) == true) .

rl [StepCase11] : GetEvent(S, Cond) Execute(Act)
                      < A : Agent | CurrentState : S, MBox : MB, AccList : ACL >
                 =>   < A : Agent | CurrentState : TargetState(S, Cond), MBox : MB, AccList : ACL > .

*******************************************Second case**************************************************************
crl [StepCase2] : Event(A, S, Cond) < A : Agent | CurrentState : S, MBox : MB, AccList : ACL >
                                        < A1 : Agent | CurrentState : S1, MBox : MB1, AccList : ACL1 >
                     => SendMessage(A : AccomplishedAction(S, Cond) : A1) GetEvent(A, S, Cond)
                         < A : Agent | CurrentState : S, MBox : MB, AccList : ACL >
                         < A1 : Agent | CurrentState : S1, MBox : MB1, AccList : ACL1 >
                     if (IsAlive(S) == false) and
                       (IsSendingAction(AccomplishedAction(S, Cond)) == true) .

rl [StepCase21] : GetEvent(A, S, Cond) SendMessage(A : Act : A1)
                      < A : Agent | CurrentState : S, MBox : MB, AccList : ACL >
                  => Event(A1, CorrespondingAgentState(S), CorrspondingCondition(Act))
                      < A : Agent | CurrentState : TargetState(S, Cond), MBox : MB, AccList : ACL >
 .
*******************************************Third case***************************************************************
crl [StepCase3] : Event(A, S, Cond)  < A : Agent | CurrentState : S, MBox : MB, AccList : ACL >
                      => ReceiveMessage(A1 : AccomplishedAction(S, Cond) : A ) GetEvent(A, S, Cond)
           < A : Agent | CurrentState : S, MBox : InsertInMBox( (A1 : AccomplishedAction(S, Cond) : A), MB),
AccList : ACL >
             if (IsAlive(S) == false) and   (IsReceivingAction(AccomplishedAction(S, Cond)) == true) .

rl [StepCase31] : GetEvent(A, S, Cond) ReceiveMessage(A1 : Act : A)
                   < A : Agent | CurrentState : S, MBox : MB, AccList : ACL >
               =>  < A : Agent | CurrentState : TargetState(S, Cond), MBox : MB, AccList : ACL > .
*******************************************Fourth case**************************************************************
crl [IniSendingToAllAct] : Event(A, S, Cond) < A : Agent | CurrentAgentState : S, MBox : MB, AccList : ACL >
                              =>
                            SendMessage(A : AccomplishedAction(S, Cond) : HeadA(ACL)) GetEvent(A, S, Cond)
                            < A : Agent | CurrentAgentState : S, MBox : MB, AccList : TailA(ACL) >
                            if (IsAlive(S) == false) and (IsSendingActionToAll(AccomplishedAction(S, Cond)) == true)
                            and (Head(ACL ) =/= Erroracquaintance) .

crl [IniSendingToAllAct] : GetEvent(A, S, Cond) SendMessage(A : Act : A1)
                               < A : Agent | CurrentAgentState : S, MBox : MB, AccList : ACL >
                              =>
                               Event(A, S, Cond)
                               < A : Agent | CurrentAgentState : S, MBox : MB, AccList : ACL >
                               Event(A1, CorrespondingAgentState(S), CorrespondingCondition(Act))
                              if Head(ACL) =/= Erroracquaintance .
endom)
```

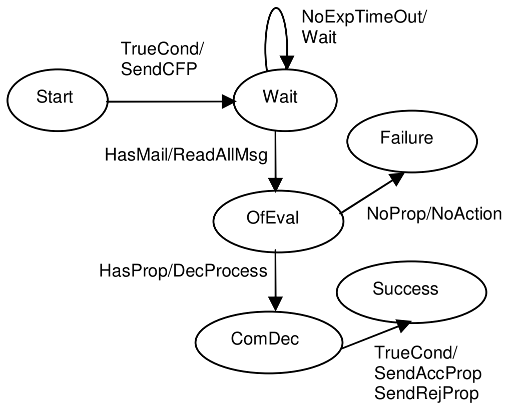
Figure 17: The object-oriented module *ATN-BASED-COMMUNICATING-AGENT*.

The meta-behavior, in the framework of our formalization approach, is expressed in four possible cases. The first case, when the action to execute is a simple internal action. In this case, the meta-behavior is described by the two rewriting rules of the first case (figure 17). One of the three remaining cases is used when it is about an action of communication. If the action of communication is a sending action, the meta-behavior is described, either by the two rewriting rules of the second case (if the destination is only one agent), or by the two rewriting rules of the fourth case (if the message is sent to all agents). The two rewriting rules of the third case describe the meta-behavior in the case where the action of communication is a reception action. Indeed, there is a serialization in the same way between every two rules. Triggering the second rule requires the execution of the first. The first rule of each part is conditional. We note that conditions are composed and the intersection of different rule conditions is the condition (*IsAlive(S) == false*). This last means that the execution of an action from a state S and requires that this state must not be final.

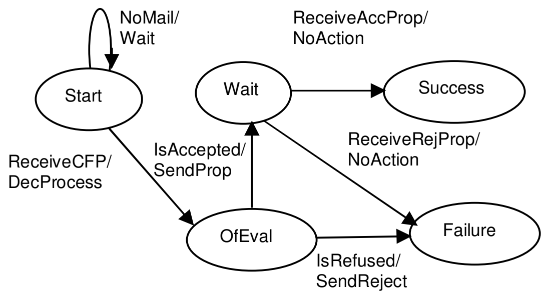
The two rules of the fourth case, present the sending of a message by the agent *A* to all its acquaintances. Such rules present a conditional loop. Indeed, they allow to browse the agent's acquaintances list *ACL*, using the two operations *HeadA* determining the element that is in head of such list, and *TailA* to determine the remainder of the list. Such a loop stops when the list is browsed completely. To every iteration, the second rule generates an event destined to the current agent (that is in head of acquaintances list ACL) corresponds to the action sent by the agent A.

6 Case Study: Auction Application

This section illustrates the application of our approach on a concrete example. It is about a simple example of an auction. We have two kinds of agents: Auctioneer and Bidder. Each auction involves one Auctioneer and several Bidders. The Auctioneer has a catalog of products. Before starting the auction, the Auctioneer sends the catalog to all the participants. Then it starts the auction for all the products. The products are therefore proposed sequentially to the participants. In this example we use the iterated contract net protocol. Figures 18.a and 18.b describe respectively the meta-behavior of the *Auctioneer*, and the *Bidder*.



(a)



(b)

Figure 18 : ATNs that describe respectively the meta-behavior of the Auctioneer and the Bidder.

6.1 Application of the Translation Process

The formal description of the behavior of an ATN-based communicating agent implies all previously defined modules with the definition of the modules *USER-ATN* and *ATN-LINK*. Figures 19 and 20 illustrate respectively the defined modules according to the meta-behaviors of the Auctioneer and the Bidder. The correspondence between the ATNs of the Auctioneer and Bidder is presented in figure 21.

```
(fmod USER-ATN1 is
extending ATN .
***definition of states constituting the ATN**********************************************************
ops StartI WaitI OfferEvaluationI CommitmentDecision SuccessI FailureI : -> NameState .      ***[1]
ops AgentState(StartI, initial)  AgentState(WaitI, ordinary)  AgentState(OfferEvaluationI, ordinary)
      AgentState(CommitmentDecisionI, ordinary) AgentState(SuccessI, final)
      AgentState(FailureI, final) : -> State .                                                 ***[2]

***definition of conditions and actions of the ATN*********************************************
ops TrueCondition NoExpiredTimeOutI HasMail HasProposal NoProposal   : -> Condition .     ***[3]
ops SendCFP WaitI ReadAllMsg  DecisionProcessI  SendAccpetProposal
      SendRejectProposal  :-> Action .                                                         ***[4]

*** determination of the target state according to a state source and a condition ***********
eq TargetState(AgentState(StartI, initial), TrueCondition) = AgentState(WaitI, ordinary) .      ***[5]
…
***   determination of the action to execute according to a state and a condition ******
eq AccomplishedAction(AgentState(StartI, initial), TrueCondition) = SendCFP .                   ***[6]
…
***Determination of the type of an action********************
eq IsInternalAction(DecisionProcessI) = true .                                                 ***[7]
…
eq IsSendingActionToAll(SendCFP) = true .                                                      ***[8]

…
endfm)
```

Figure 19: The module USE-ATN corresponding to the agent Auctioneer

In figure 19, we define the different states of the agent *Auctioneer* (lines [1, 2]). For example, the state *AgentState(StartI, Initial)* means that the state named *StartI* is an initial state (see figure 18.a). The conditions and actions given in figure 18.a are described respectively by (line [3, 4]). To determine the target state (line [5]) according to a source state and a given condition, we used the operation *TargetState* defined in figure 10. If the agent *Auctionner* is in its initial state *StartI*, and the condition *TrueCondition* is satisfied, the target state of this transition must be the ordinary state *WaitI*. The accomplished action depends on a source state and a condition. *SendCFP* is the action according to the condition *TrueCondition* when the *Auctioneer* is in its initial state *StartI* (line [6]). To select the conditional rule (see figure 17) to be executed, it's necessary to know the action type. For example, line [7] indicates that the action *DecisionProcessI* is an internal action, and line[8] indicate that the action *SendCFP* must be sent by the *Auctioneer* to all *Bidders*.

In the agent community, interaction protocols play a primordial role. They facilitate the coordination between agents to achieve their goals. To respect the used protocol, we propose to establish a correspondence between states of agents (see figure 21).

```
(fmod USER-ATN2 is
extending ATN .
***definition of states constituting the ATN***********************************************************
ops StartP WaitP OfferEvaluationP SuccessP FailureP : -> NameState .
ops AgentState(StartP, initial)  AgentState(WaitP, ordinary)  AgentState(OfferEvaluationP, ordinary)
      AgentState(SuccessP, final) AgentState(FailureP, final) : -> State .

***definition of conditions and actions of the ATN*********************************************
ops ReceiveCFP NoMail IsAccepted IsRefused ReceiveAcceptProposal   ReceiveRejectProposal : -> Condition
ops DecisionProcessP WaitP SendPropose RejectPropose   :-> Action .
*** determination of the target state according to a state source and a condition ***********
eq TargetState(AgentState(StartP, initial), ReceiveCFP) = AgentState(OfferEvaluationP, ordinary) .
…
***   determination of the action to execute according to a state and a condition ******
eq AccomplishedAction(AgentState(StartP, initial), ReceiveCFP) = DecisionProcessP .
…
***Determination of the type of an action********************
eq IsInternalAction(DecisionProcessP) = true .
eq IsSendingAction(SendPropose) = true .
…
endfm)
```

Figure 20 : The module USE-ATN corresponding to the agent Bidder.

For example, if the agent *Auctioneer* is in its initial state *StarI*, a *Bidder* must be in its initial state *StartP* (line[1]). In the same way, if the *Bidder* is in its ordinary state *OfferEvaluation* (line [3]), the Auctioneer must wait its decision. Indeed, the conditions of transitions described in an ATN test the occurrence of an asynchronous event (message reception, new data, …) or an internal state change. Therefore, a sending action of a sender agent corresponds to an event for the agent receiver. For example, when the *Auctioneer* launches a call-for–proposal (*SendCFP*), the agent *Bidder* receives the event call-for-proposal (*ReceiveCFP*). This is expressed by the equation of line [2].

```
fmod ATN-LINK is
protecting USER-ATN1 .
protecting USER-ATN2 .

op CorrespondingAgentState : State -> State .
op CorrespondingCondition : Action -> Condition .

************** Auctionner part ******************
eq CorrespondingState(AgentState(startI, initial)) = AgentState(startP, initial) .                    ***[1]
eq CorrespondingState(AgentState(commitmentdecisionI, ordinary)) = AgentState(waitP, ordinary) .
...
eq CorrespondingCondition(SendCFP) = ReceiveCFP .                                                     ***[2]
eq CorrespondingCondition(SendAcceptProposal) = ReceiveAcceptProposal .
...

************** Bidder part ******************
eq CorrespondingState(AgentState(waitP, ordinary)) = AgentState(offerEvaluationI, ordinary) .        ***[3]
eq CorrespondingState(AgentState(offerEvaluationP, ordinary)) = AgentState(waitI, ordinary) .
...
eq CorrespondingCondition(SendPropose) = HasProposal .                                                ***[4]

eq CorrespondingCondition(SendReject) = NoProposal .
...
endfm
```

Figure 21 : The module ATN-LINK .

6.2 Validation of the Generated Description

The rewriting logic offers a great flexibility in terms of simulation of a specification, in particular, concerning the choice of the initial configuration. This choice plays a primordial role in the validation of the description of a system. Using all the system’s description, we can validate a part of the system without involving the rest. We worked on the simulation of several features related to the dynamic behavior of the agents. We also consider: (1) the behavior that starts by sending a CFP message by the Auctioneer to the Bidders and finishes by a success after sending a message of accept proposal to one of the Bidders, and (2) the behavior that begins by sending a CFP message by the Auctioneer to the Bidders, and finishes by a failure after Bidders refuse to propose. We take an example (Figure 22) of the first case with a given initial configuration. The unlimited rewriting process (without indicating the number of rewriting steps) of this configuration gives the result illustrated in figure 23.

```
< "Auctioneer" : Agent | CurrentAgentState : AgentState(startI, initial), MBox : EmptyMailBox, AccList : “Bidder” >
< "Bidder" : Agent | CurrentAgentState : AgentState(startP, initial), MBox : MB1, AccList : “Auctioneer” >
Event("Auctioneer", AgentState(startI, initial), TrueCondition)
Event("Bidder", AgentState(offerEvaluationP, ordinary), IsAccepted)
Event("Auctioneer", AgentState(waitI, ordinary), HasMail)
Event("Auctioneer", AgentState(commitmentdecisionI, ordinary), TrueCondition) .
```

Figure 22: initial configuration

```
< "Auctioneer" : Agent | CurrentAgentState : AgentState(successI, final), MBox : EmptyMailBox, AccList : “Bidder” >
< "Bidder" : Agent | CurrentAgentState : AgentState(successP, final), MBox : MB1, AccList : “Actioneer” >
```

Figure 23: Auctioneer and Bidder finish in success.

The Auctioneer sends a CFP to the Bidders. After evaluating the proposals, the Auctionner chooses one of the Bidders, by sending it a message of accept proposal. This last receives the message and finishes by a success. To show the second case, we choose the initial configuration of figure 24. The unlimited rewriting of this configuration gives the result illustrated by figure 25.

```
< "Auctioneer" : Agent | CurrentAgentState : AgentState(startI, initial), MBox : EmptyMailBox, AccList : “Bidder” >
< "Bidder" : Agent | CurrentAgentState : AgentState(startP, initial), MBox : MB1, AccList : “Auctioneer” >
Event("Auctioneer", AgentState(startI, initial), TrueCondition)
Event("Bidder", AgentState(offerEvaluationP, ordinary), IsRefused)
Event("Auctioneer", AgentState(waitI, ordinary), HasMail) .
```

Figure 24: initial configuration

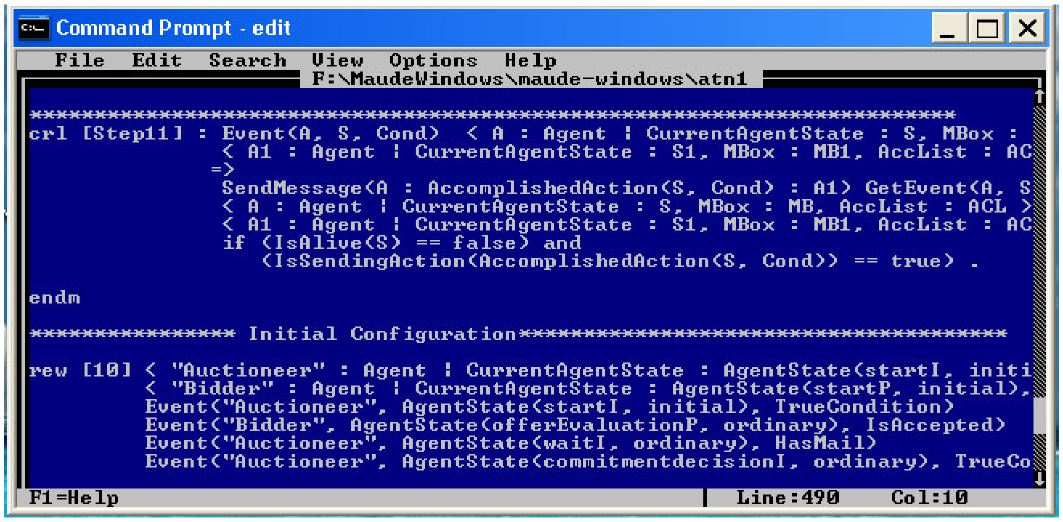
```
< "Auctioneer" : Agent | CurrentAgentState : AgentState(FailureI, final), MBox : EmptyMailBox, AccList : ACL >
< "Bidder" : Agent | CurrentAgentState : AgentState(FailureP, final), MBox : MB1, AccList : ACL1 >
```

Figure 25: Auctioneer and Bidder finish in Failure.

The auctioneer sends a CFP and launches its process of evaluation after a given deadline. It finds no proposition. Therefore, all agents finish in failure.

6.2 Implementation

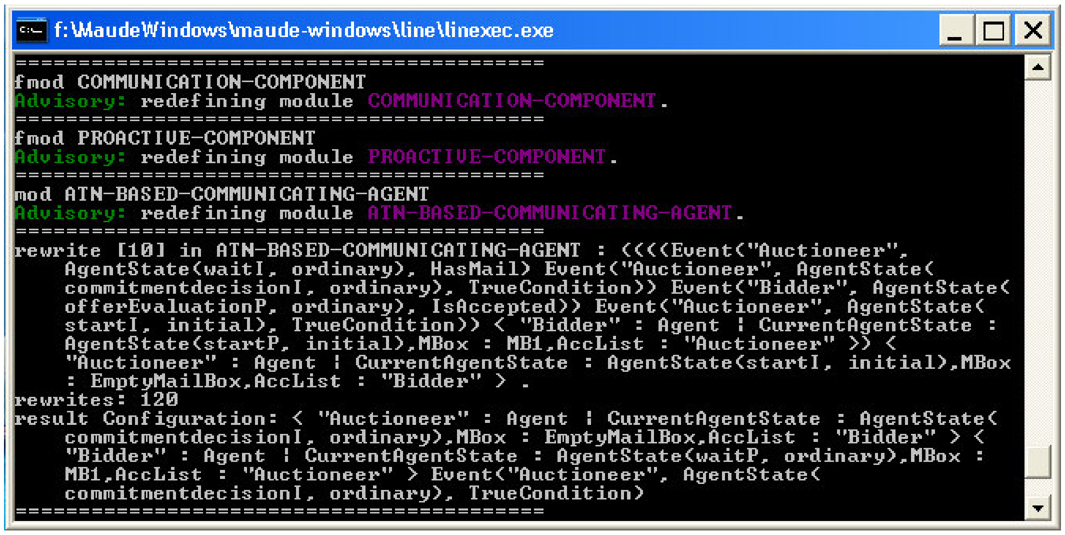
Figure 26 illustrates a part of the code we developed. It visualizes, on the one hand, the rewriting rule that describes the agent's behavior in the case where the final state would not even extinct and, on the other hand, the limited rewriting (after 10 steps) of the same initial configuration illustrated by figure 22.


Command Prompt - edit
File Edit Search View Options Help
F:\MaudeWindows\maude-windows\atn1
F1=Help
Line:490 Col:10

```
**********************************************************************
crl [Step11] : Event(A, S, Cond)  < A : Agent | CurrentAgentState : S, MBox :
               < A1 : Agent | CurrentAgentState : S1, MBox : MB1, AccList : AC
              =>
               SendMessage(A : AccomplishedAction(S, Cond) : A1) GetEvent(A, S
               < A : Agent | CurrentAgentState : S, MBox : MB, AccList : ACL >
               < A1 : Agent | CurrentAgentState : S1, MBox : MB1, AccList : AC
               if (IsAlive(S) == false) and
                  (IsSendingAction(AccomplishedAction(S, Cond)) == true) .

endm

**************** Initial Configuration**************************************

rew [10] < "Auctioneer" : Agent | CurrentAgentState : AgentState(startI, initi
         < "Bidder" : Agent | CurrentAgentState : AgentState(startP, initial),
         Event("Auctioneer", AgentState(startI, initial), TrueCondition)
         Event("Bidder", AgentState(offerEvaluationP, ordinary), IsAccepted)
         Event("Auctioneer", AgentState(waitI, ordinary), HasMail)
         Event("Auctioneer", AgentState(commitmentdecisionI, ordinary), TrueCo
```

Figure 26: Part of the developed code.

The result of limited rewriting of the initial configuration indicated in figure 26 is illustrated by figure 27. The Auctioneer launches its process of decision and the Bidder waits its response.



```
==========================================
fmod COMMUNICATION-COMPONENT
Advisory: redefining module COMMUNICATION-COMPONENT.
==========================================
fmod PROACTIVE-COMPONENT
Advisory: redefining module PROACTIVE-COMPONENT.
==========================================
mod ATN-BASED-COMMUNICATING-AGENT
Advisory: redefining module ATN-BASED-COMMUNICATING-AGENT.
==========================================
rewrite [10] in ATN-BASED-COMMUNICATING-AGENT : ((((Event("Auctioneer",
    AgentState(waitI, ordinary), HasMail) Event("Auctioneer", AgentState(
    commitmentdecisionI, ordinary), TrueCondition)) Event("Bidder", AgentState(
    offerEvaluationP, ordinary), IsAccepted)) Event("Auctioneer", AgentState(
    startI, initial), TrueCondition)) < "Bidder" : Agent | CurrentAgentState :
    AgentState(startP, initial),MBox : MB1,AccList : "Auctioneer" >) <
    "Auctioneer" : Agent | CurrentAgentState : AgentState(startI, initial),MBox
    : EmptyMailBox,AccList : "Bidder" > .
rewrites: 120
result Configuration: < "Auctioneer" : Agent | CurrentAgentState : AgentState(
    commitmentdecisionI, ordinary),MBox : EmptyMailBox,AccList : "Bidder" > <
    "Bidder" : Agent | CurrentAgentState : AgentState(waitP, ordinary),MBox :
    MB1,AccList : "Auctioneer" > Event("Auctioneer", AgentState(
    commitmentdecisionI, ordinary), TrueCondition)
==========================================
```

Figure 27 : Intermediate result of the limited rewriting process (after 10 steps) of the initial configuration.

7 Conclusions and Future Work

Formal specification is an important issue in MAS development. Several approaches addressing the specification of multi-agent's behavior have been proposed during the last years. They are based on different formalisms (statechart, Petri Net or AUML). Such formalisms adopt, in general, a diagrammatic or semi-formal notations. Furthermore, such approaches only offer a partial formalization of multi-agents systems. Using formal notations to specify MAS' behavior makes it possible to produce precise descriptions. This also offers a better support to their verification and validation process. In this paper, we presented a formal framework called DIMA-Maude for supporting the formal description and validation of DIMA models. The framework is described using the formal language Maude. The developed framework, in its present version, is extensible. It is composed of several modules representing the basic bricks for the formal description and validation of DIMA communicating agents whose meta-behavior is described by an ATN. The Maude language is supported by a tool. It allowed us, in addition of modeling, validating (based on a simulation) our approach. Moreover, our model is reusable. This reusability is expressed essentially by the decomposition that we performed. Furthermore, we plan to extend our approach by integrating possibilities offered by the Maude language (model-checker) to verify some properties of the specification of the DIMA model described in Maude. In this future direction, we will verify, on the one hand, the internal behavior of a DIMA agent and, on the other hand, the collective behavior of a group of DIMA agents.

8 References

[Bak00] I. Bakam, F. Kordon, C. Le Page, F. Bousquet. “Formalization of a Spatialized Multiagent Model Using Coloured Petri Nets for the Study of a Hunting Management System”. First International Workshop, FAABS 2000, Greenbelt, MD, USA, April 2000. FAABS 2000.

[Bet92] M. Bettaz, M. Maouche. “How to specify Non Determinism and True Concurrency with Algebraic Term Nets”. Lecture Notes in Computer Science, N 655, Spring Verlag, Berlin, p. 11-30, 1992.

[Cha01] A. Chaoui, M. Bouzenada. “G-ECATNets: An Object Petri Net-Based Framework for the Modular Design of Complex Information Systems”. ISIICT’2001, 2001.

[Cos99] R. Cost and al. “Modeling Agent Conversations with colored Petri Nets”, à paraître dans Working Notes of the Workshop on Specifing and Implementing Conversation Policies, Autonomous Agents’99, seattle, Washington, mai 1999.

[Fra00] N. Franchesquin, B. Espinasse. “Analyse multi-agents de la gestion hydraulique de la camangue : considérations méthodologiques”. Communication soumise à JFIADSMA’2000, 2-4 oct 2000 Saint Etienne, 2000.

[Gue96] Z. Guessoum. "Un environnement opérationnel de conception et de réalisation de systèmes multi-agents". Thèse de l'Université Paris 6, LAFORIA, mai 1996.

[Gue03] Z. Guessoum. "Modèles et Architéctures d'Agents et de Systèmes Multi-Agents Adaptatifs". Dossier d'habilitation à diriger des recherches de l'Université Pierre et Marie Curie. Décembre 2003.

[Man99] M. Clavel and al. "Maude : Specification and Programming in Rewriting Logic". Internal report, SRI International, 1999.

[Man03] M. Clavel and al. "Maude 2.0 Manual, Version 1.0". Internal report, SRI International, june 2003.

[McC03] T. McCombs. "Maude 2.0 Primer, Version 1.0". Internal report, SRI International, 2003.

[Mes90] J. Meseguer, "Rewriting as a unified model of concurrency". In Proceedings of the Concur'90 Conference, Amsterdam, Pg 384-400, Springer LNCS Vol. 458, 1990.

[Mes92] J. Meseguer, "A Logical Theory of Concurrent Objects and its Realization in the Maude Language". In G. Agha, P. Wegner, and A. Yonezawa, Editors, Research Directions in Object-Based Concurrency. MIT Press, 1992.

[Ode00] J. Odell, H. V. D. Parunak, B.Bauer, "Representing agent Interaction protocol In UML", conférence AAAI Agents 2000, Barcelone, 3-7 juin 2000.

[Ode01] J. Odell, H. V. D. Parunak, B.Bauer, "Representing agent Interaction protocol In UML", Agent Oriented Software Enginering, Paolo Ciancarini and Michael Wooldridge (eds.), Springer-Verlag, Berlin, 2001, pp. 121-140.

[Pau02] S. Paurobally. "Rational Agents and the Processes and States of Negotiation", Imperial College , Ph.D, Thesis 2002.

[Pau03] S. Paurobally, J. Cunningham, "Achieving Common Interaction Protocols in Open Agent Environments", 2nd international workshop on Challenges in Open Agent Environments, AAMAS 2003, Melbourne, Australia 14-18th July 2003.

[Pen99] Y. Peng and al. "An agent based approach for manufacturing integration_ the CI-IMPLEX experience" . International Joural of Applied Artificial Intelligence, 13(1-2):39-64, 1999.

[Saa93] G. Saake, T. Hartman, R. Junglaus, H-D. Ehrich, "Object-Oriented Design of Information Systems: Troll language Features". In Procedings CISM School Udine'93, LNCS, Springer Verlag, 1993.

[Sco98] R. Scott Cost and al. Jackal: ''A Java-based tool for agent development''. In Jeremy Baxter and Chairs Brian Logan, editors, Working Notes of the WorkShop on Tools for Developing Agents, AAAI'98, number WS-98-10 in AAAI Technical Reports, pages 73-82, Minneapolis, July 1998. AAAI, AAAI Press.

[Tim97] Tim Finin and al. "KQML as an agent communicating language". In Jeff Bradshaw, editor, software Agents. MIT Press, 1997.

[Tra99] E. Tranvouez, B. Espinasse, "Protocoles de coopération pour le réordonnancement d'atelier". in actes des journées francophones d'Intelligence Artificielle Distribuée et Systèmes Multi-Agents (JFIADSMA'99) à Saint-Gilles, île de la Réunion, novembre 1999, Gleizes J.-P., Marcenac P., Ed. Hermès, 1999.