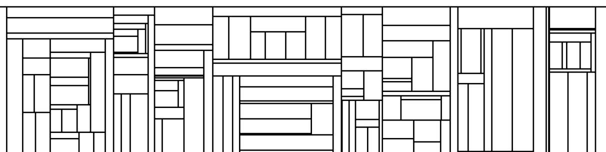
ISSN 0105-8517

Second Workshop on Modelling of Objects, Components and Agents Aarhus, Denmark, August 26-27, 2002

Daniel Moldt (Ed.)

DAIMI PB - 561
August 2002

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark



Preface

This report contains the proceedings of the workshop Modelling of Objects, Components, an Agents (MOCA'02), August 26-27, 2002. 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/workshop02/

Objects, components, and agents as fundamental concepts are 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 programme committee that selected the papers consists of:

Wil van der Aalst	The Netherlands
Remi Bastide	France
Jonathan Billington	Australia
Didier Buchs	Switzerland
Henrik Bærbak Christensen	Denmark
Jose-Manuel Colom	Spain
Jörg Desel	Germany
Susanna Donatelli	Italy
Nisse Husberg	Finland
Jens Bæk Jørgensen	Denmark
Francisco José Camargo Santacruz	México
Ekkart Kindler	Germany
Gabriela Kotsis	Austria
Fabrice Kordon	France
Sadatoshi Kumagai	Japan
Rainer Mackenthun	Germany
Daniel Moldt (Chair)	Germany
Tadao Murata	USA
Dan Simpson	United Kingdom
Rüdiger Valk	Germany
Tomas Vojnar	Czech Republic
Wlodek M. Zuberek	Canada

The programme committee has accepted 8 papers for presentation. They tackle the concepts of objects, components, and agents from different perspectives. Formal as well as application aspects demonstrate the wide range within which Petri nets can be used. At the same time they illustrate that there is a tendency to use more high-level concepts for the analysis and design of Petri-net-based models.

Daniel Moldt

Reviewers

The organisers of MOCA'02 would like to express their appreciation for the work of the reviewers listed below.

Wil van der Aalst	Eindhoven University of Technology (EUT), Netherlands
Remi Bastide	University of Toulouse, France
Jonathan Billington	University of South Australia, Australia
Didier Buchs	Swiss Federal Institute of Technology Lausanne, Switzerland
Henrik Bærbak Christensen	University of Aarhus, Denmark
Jose-Manuel Colom	University of Zaragoza, Spain
Jörg Desel	Catholic University of Eichstätt-Ingolstadt, Germany
Susanna Donatelli	University of Torino, Italy
Nisse Husberg	Helsinki University of Technology, Finland
Jens Bæk Jørgensen	University of Aarhus, Denmark
Francisco José Camargo Santacruz	Technological Education of State of Mexico, México
Ekkart Kindler	University of Paderborn, Germany
Michael Köhler	University of Hamburg, Germany
Fabrice Kordon	University Pierre an Marie Currie of Paris, France
Gabriela Kotsis	University of Vienna, Austria
Sadatoshi Kumagai	University of Osaka, Japan
Gabriela Lindemann	Humboldt University Berlin, Germany
Rainer Mackenthun	Fraunhofer Institute for Software and Systems Engineering, Germany
Tadao Murata	University of Illinois at Chicago, USA
Heiko Rölke	University of Hamburg, Germany
Dan Simpson	University of Brighton, UK
Mark-Oliver Stehr	University of Hamburg, Germany
Rüdiger Valk	University of Hamburg, Germany
Tomas Vojnar	Technical University of Brno, Czech Republic
Klaus Voss	German National Research Center for Information Technology (GMD), Germany
Bin YuNorth	Carolina State University, USA
Wlodek M. Zuberek	Memorial University of Newfoundland, Canada

Table of Contents

Invited Talk:
Hans-Dieter Burkhard
Software-Architectures for Agents and Mobile Robots..................................... 1

Mao Xinjun, Wu Gang, Wang Huaimin, and Zhao Jianming
Formal Model of Joint Achievement Intention.. 19

H. Djenidi, A. Ramdane-Cherif, C. Tadj and N. Levy
Generic Multi-Agent Architectures for Multimedia Multimodal Dialogs........ 29

F. Franceschinis, M. Gribaudo, M.Iacono, N. Mazzocca, and V. Vittorini
Towards an Object Based Multi-Formalism Multi-Solution Modeling
Approach... 47

Jukka Järvenpää and Marko Mäkelä
Towards Automated Checking of Component-Oriented Enterprise
Applications 67

Invited Talk:
Søren Christensen
Moddeling with Coloured Petri Nets 87

Jens Bæk Jørgensen and Claus Bossen
Executable Use Cases for Pervasive Healthcare.. 89

W.M.P van der Aalst
Inheritance of Dynamic Behaviour in UML.. 105

Danny Weyns and Tom Holvoet
A Coloured Petri Net for a Multi Agent Application 121

Michael Köhler and Heiko Rölke
Modelling Mobility and Mobile Agents using Nets within Nets................... 141

Software-Architectures for Agents and Mobile Robots

Hans-Dieter Burkhard
Institute of Informatics
Humboldt University
Berlin, Germany
hdb@informatik.hu-berlin.de

Abstract

Agents and mobile robots are implemented to act "autonomously on behalf of their user/owner". They have to interact with virtual or real-world environments. This leads to a first "horizontal" modularization according to perception, control, and actuation. Reactive behavior is implemented by simple translations from sensors to actuators, deliberative behavior includes complex goal selection and planning. Hybrid architectures combine both approaches using layered architectures, which leads to a second vertical modularization. The synchronization and interaction between the modules poses serious problems when the agents/robots have to work on complex tasks in dynamic environments. Persistent states are used to maintain past oriented and future oriented information: The world model combines new perceptions with previous ones, and the commitment maintains plans for the achievements of long term goals. Special efforts are needed to keep balance between stabile behavior and adaptation to new situations. The implementation of "bounded rationality" needs new architectures behind the scope of the classical ones.

1 Introduction

Control of autonomous robots in dynamical environments is interesting from a cognitive point of view as well as under application view points. Technical requirements are established to construct intelligent autonomous systems in virtual worlds like the internet as well as in the real world. But still there is an ongoing debate about the best way to control intelligent behavior. Examples from nature include

1. Immediate reactions to inputs from the real world [Maes90], [Brooks91]:
   This approach can lead to surprisingly complex behavior if the stimulus-response actions are well tuned. The basic idea behind this approach is to use the complexity of the environment for control: The best model of the world is the world itself, complex behavior *emerges* from the interaction with the world. But note that virtual reality worlds must simulate the physical relations including substitutes for body sensors to a very detailed level to allow efficient stimulus-response behaviors.

2. Actions following long term plans [Bratman87], [Rao/Georgeff91]:
   The control uses complex internal models which are analyzed for reachable goals. Plans are developed to achieve the goals. It needs a lot of efforts to make appropriate models even for simple behaviors like following a path while avoiding obstacles in a dynamically changing environment. But complex behavior (e.g. playing chess or constructing an air plan) needs a lot of appropriate proactivity.

3. Swarm intelligence [Parunak97]: Complex behavior emerges from the interaction of large groups of simple agents. This approach can be seen as an extension of the first one using cooperation. Cooperation emerges by similar reactions of the agents to similar sensory inputs. Moreover, the results of the activities in the world are used as stimuli of other agents (e.g. the use of chemical substances as markers by insects). Flexibility and adaptation are realized by a certain randomness of actions.

The paper is concerned with individual agents, i.e. with the first two approaches. Layered architectures are used for combination [Arkin98, Murphy00]: Lower layers implement fast reactions using "behaviors", higher layers implement the guidance of behaviors by plans. The higher layers are called with lower frequencies and have longer reaction times.
Different behavior needs different triggers. Simple stimulus-response behavior is triggered by recent sensory data only. But often the environment does not provide appropriate sensory data for triggering the achievement of a long term goal (e.g. running to intercept a ball coming from behind - this point will be discussed in more detail below). This means the agent needs some knowledge about the situation in the outside world behind the sensory data and some knowledge about its goals and plans.
More abstractly spoken, the agent possesses "persistent" (mental) states to memorize world models and goals, respectively. We call them "persistent" states to emphasize their *persistence over longer time intervals*. This is necessary to make a clear distinction concerning certain software aspects: During complex computation processes we have "intermediate" states. Often used methods for action selection are decision trees, state machines, rule bases etc. These methods may go through different "intermediate" states while performing the selection process. But these intermediate states are forgotten when the process is finished. But if the result of the process needs to be stored (like a goal to be achieved in the future), then this is realized using a persistent state. It is used as an internal trigger for forthcoming decision processes.
Commonly used notions are "reactive" and "deliberative" behavior, respectively. Reactive behavior is mostly understood as simple behavior, without (persistent) commitment to goals and plans. Deliberative behavior is identified with complex decisions. There are different aspects mixed in these notions as

- complexity of the decision process,
- ability to anticipate possible future developments,
- planning capabilities,
- persistent states concerning the past (persistent worldmodel),
- persistent states concerning the future (persistent commitments).

As an example we may consider a chess program: It can anticipate future situations considering the possible moves of both players starting with the recent situation. It can evaluate reachable future "goals" using complicated evaluation procedures. Finally it comes up with simply the next move, and all intermediate results are forgotten. After the opponent's move, the same process starts again for the new situation without any reference to the previous computations. Is it a reactive behavior? We cannot solve these terminological problems in this paper, but we will discuss some of its aspects and the reasons behind.
The practical problems concern rapid reactions to fast changes in the environment. Reactive behaviors are considered appropriate for rapid reactions, while deliberative ones are

concerned with long term planning. If long term planning is used in dynamically changing environments, both approaches are needed, but then they are in conflict concerning their synchronization. Layered architectures combine deliberative ”higher” layers with reactive ”lower” layers. Different synchronization strategies are in use, but usually the higher layers have some delay because they are computationally expensive. Hence only lower reactive layers react dynamically. Thereby they act according to the long term goals defined by the higher deliberative layers. But this goals remain the old ones as long as there is no redeliberation regarding the new requirements. In fact, there is a real time control problem concerning fast redeliberations.
To allow some kind of short term redeliberation in complex environments, it is inevitably to restrict the search space for rapid decisions. This corresponds to concepts of bounded rationality, where a special ”screen of admissibility” ([Bratman87]) is introduced for the restriction of deliberation processes. The proposal in this paper is a new architecture with two separated passes through all levels of control as an attempt to combine complex long term decisions with short term behaviors under real time conditions.
The paper is organized as follows: General aspects of robot controls in dynamical environments are considered in Section 2 using the scenario of soccer playing robots (RoboCup). Control architectures are discussed in Section 3. Section 4 continues the discussion of control problems, the impacts of these problems to the design of control architectures are investigated. This analysis leads to the proposal of an hierarchically structured control architecture in Section 5. It allows for long term and short term decisions on all levels of the hierarchy. In contrast to other layered architectures, just-in-time decisions are possible on the higher levels, too. An extended version of the paper will appear in Fundamenta Informaticae [Burkhard02].
The author likes to thank the members of the teams ”AT Humboldt” and ”German Team” in the RoboCup for a lot of fruitful discussions. The work is granted by the German Research Association (DFG) in the research program 1125 ”Cooperating teams of mobile robots in dynamic and competitive environments”.

2 Robot Control in Dynamic Environments

Dynamic environments are characterized by fast changes, such that plans may become invalid by unpredictable events. The robot football (European ”football”, i.e. ”soccer”) scenario promoted by the RoboCup initiative [RoboCup] [Kitano-et-al-97] is best suited as an illustrative example. It provides a dynamic environment for the football/soccer playing robots. Special characteristics are the presence of adversaries and the availability of only incomplete, imprecise data. One may theoretically think about a plan to play the ball via several players from the goal-kick to the opponents goal, but nobody would expect that plan to work. Note that there is a great difference to a chess program: It is easy to write a program for finding the ultimate best moves, it is ”only” a question of complexity to run this program. But nobody is able to write a similar program for football/soccer playing robots.
It is important to realize that the robots have to work autonomously without any outside control. Moreover, there is no global control in our scenario: Each robot has to decide for its own with restricted knowledge about the environment and about other robots. Some communication is provided, but not enough to exchange detailed information about the situation and about decisions (the amount of data is restricted).
Control structures for intelligent robots/agents include

- sensors and perception unit to get inputs from the environment,

- behavior control (with different complexities ranging from simple stimulus response behavior to long term deliberative behavior as discussed in this paper),
- actors and basic action control to act in the environment (sometimes using direct feed back with sensors),
- operating system for synchronizing the different activities (using parallel processing if possible).

Communication capabilities are included in the sensors and actors, respectively. There exist a lot of different approaches for controls of intelligent agents and intelligent robots (cf. e.g. [Arkin98, Murphy00, Weiß99]).

2.1 Basic Skills

The football/soccer scenario provides a lot of different situations to illustrate the needs of agent architectures for dynamic environments. They range from basic skills up to complex cooperative behavior.

- The raw input information provided by sensors is processed to yield a perception. The resulting data structure models the environment including the robot itself (especially positions and movements of the ball and of the players). It is called the "worldmodel".
- The information provided by sensors in a single moment is incomplete (the ball may be covered by other players) and imprecise (due to noisy data). It is possible to build a more complete worldmodel using information from the past together with the new perception. For example, the movement of a ball which is covered by other players can be anticipated using information from the old worldmodel.

  The important aspect of such a worldmodel is its *persistence* with respect to the time scale induced by sensor inputs and effector outputs. The agent maintains such a worldmodel as a *persistent state* with updates according to new sensory information. Since it is oriented to information from the past, it is called past-oriented mental state.
- Interception of a moving ball illustrates simple problems of the dynamic environment: A very simple "stimulus-response player" would run straight line to the place where he sees the ball. As the ball is moving he has to adjust its direction every time he looks for the ball, and he will perform a curved path as the result. A more skillful player could anticipate the optimal point for interception and run directly to this point.
- Now we discuss such a procedure for the anticipation of the optimal point for interception. It calculates the speed vector v for the optimal run to the ball depending on the recent position p and the speed u of the ball (relative to the player). It may use additional parameters according to opponents, whether conditions, noise etc.

  The calculation may explicitly exploit physical laws (including e.g. the expected delay of the ball). It may use simulation (forward model) for possible speed vectors v of the player. If an inverse model is available, the optimal speed vector v may be calculated directly. Calculations of v may use a neural network which has been trained by real or simulated data. (Which of these methods should be called "reactive"?)

- We still consider the optimal interception of a moving ball using calculations of the speed vector v. The calculation can be repeated whenever new sensor information is available. Therewith it always can regard newest information and hopefully obtain the best speed vector v. Alternatively, the player may keep moving according to v for a longer time. Therefore he needs another kind of *persistent state* to memorize this goal. Since this state is oriented to information concerning the future, it may be called future-oriented mental state. It can save computation time, and it is useful to keep stable behavior (see below).

  If the ball is not observable for some time (e.g., if it is covered by another player), then the persistent goal is used as the trigger to keep running. Alternatively, simulating the ball in the worldmodel can also be a trigger to continue the interception process.

- Problems with the reliability of the computed speed vector v arise due to noise in the sensory data (and may be due to imprecise calculations themselves). Repeated calculations may hence result in oscillations and sub-optimal behavior (as reported e.g. in [Müller-Gugenberger/Wendler98]). It may be better to follow the old speed v_t as long as the difference to the new speed v_{t+1} is not too large. Keeping v_t in a future oriented mental state provides the necessary means. Exploiting the inertia of the robot provides another way using the physical world directly. A complete analysis of the problems behind stability and flexibility go behind the scope of this paper (cf. e.g. [Bratman87],[Burkhard00]).

The discussion shows a lot of different approaches and implementations for the simple behavior "follow a moving object". In most cases there is a lot of redundancies which can be exploited for efficient and more reliable controls in different ways. It is a typical observation in robot control that the same behavior can be realized in different ways yielding different trade offs. Since single methods are often of restricted reliability, the appropriate combination (regarding the overall system) is a challenging design problem.
To summarize: Two concepts of persistent ("mental") states have been introduced. It is commonly accepted that some form of persistent state is essential even for primitive beings. The worldmodel as a persistent state concerning the past compensates missing sensory information from the outside world. The persistent state concerning the future may be not really necessary at the level of basic skills.

2.2 Coordination

More complex problems of dynamic environments are illustrated by coordination. The decision processes become more and more complex (and subject to stability problems) as the time horizon is enlarged. Even in the recent Simulation League of RoboCup (the competitions in a virtual environment which do not suffer from the physical robot problems), a coordinated behavior like a double pass emerges only sometimes by chance, not by planned activities. Here are some examples of decision processes in the RoboCup scenario:

- A player decides if he can intercept the ball, i.e. if the ball is reachable during its move on the playground. The decision process can use the procedures for computing v from above to calculate the interception point and time.

- A player decides if he can intercept the ball before any other player. Therefore he has to compare his own chances with the interception times of other players (e.g. using the methods to calculate v from the view point of other players).

- A player decides not to intercept the ball even if he is the first to reach the ball. The reason may be a team mate in a better position for continuation.

Next we had to discuss the optimal behavior for all the players which are not in a position to control the ball directly. Their optimal behavior is determined by long term strategic items, and it is important for the success of a robot team. Humans often use predefined behavior patterns for coordination, like change of wings, double pass etc. in football/soccer. The reader is invited to think about the related problems as discussed above for interception: maintenance of information from the past, anticipation of future chances, managing of stability and optimality, – all under the conditions of dynamic changes and incomplete imprecise information. There is a growing value of global (symbolic) descriptions of situations and behaviors in order to guide short term behavior by long term goals. Goals and plans are memorized by future oriented persistent states for at least two reasons, namely efficiency (repeated computations should be avoided) and stability (needed for cooperation of team mates).

3 Architecture Models

3.1 A Simple State Model

The notions of persistent states are discussed somewhat more formally in this section. A discrete control of the agent is considered for the sake of simplicity. There is some freedom for choosing the time steps $t = 0, 1, 2, \dots$. There are good reasons to identify the time steps with the arrival of sensory data ("input") at the control unit (e.g. we have some kind of event driven control). Note that persistence depends on this definition: We call a state a persistent state if and only if it keeps information from one step to the next. The chess program considered in Section 1 does not have persistent states. It needs no persistent worldmodel if all board positions are used as input, and it needs no memorizing of goals if evaluation of possible moves starts from scratch for the new situation.
Computed goals and plans of a football/soccer robot are not persistent as long as they cannot be used by the decision process in the next time step. (Our implementation of the control architecture in [Burkhard *et al.*98] was based on the notions of belief, desire and intentions to describe intermediate results. Actually, the concepts did not stand for persistent states since the decision process was started from the beginning in each time step. But then certain stability problems occurred like oscillating directions while intercepting the ball. They were solved by references to old intentions later.)
A generic agent-oriented control architecture with a simple cyclic process ("sense-think-act"-cycle) is widely used. The cycle is performed at each time step t.

1. Input (sense): Data have been collected by the agent. They may come from outside (sensors), via communication and by body information (proprioceptive sensors). The data is preprocessed yielding some internal representation of the environment which is called "worldmodel". (Here the notion "worldmodel" may stand for non-persistent data, too.)

2. Commit (think): The control unit analyses the worldmodel. It may evaluate possible courses of actions and possible future situations. It commits for actions to be performed immediately and perhaps for long term behavior.

3. Output (act): The control unit outputs the advice for actions to be performed by the agent immediately. The actions are performed (may be after some further processing) by effectors, and by communicators.

The most simple architecture is an architecture without any persistent state as in Figure 1. Only the most recent input can be used for the control. This causes no problems if the input is complete and reliable as far as necessary for commitments.

```
for t=0,1,2,...  do
      worldmodel_t := perceive(input_t);
      commitment_t := deliberate(worldmodel_t);
      output_t := execute(commitment_t);
```

Figure 1: Stimulus-response Architecture without Persistent World model

The deliberate-function can be a simple table, a neural network or a complicated decision process using goals and plans – remember the chess program. But the commitments are used only for the recent output, then they are completely forgotten in the case of stimulus-response architectures.

Next the stimulus-response architecture with persistent world model is considered as in Figure 2. A persistent worldmodel allows to regard former inputs. The agent can try to maintain a complete worldmodel even if the most recent sensor information is incomplete. The new input is integrated into the existing worldmodel, missing facts can be simulated to some extend. This means that the agent has the ability to anticipate world states. It is common understanding that the persistent worldmodel is used only as a "past-oriented" state which memorizes information concerning the situation of the outside world: It serves as a substitute for a complete and precise sensory information by the input. Again the deliberate-function can be very simple or complex, respectively. Commitments are used only for the recent output.

```
for t=0,1,2,...  do
      worldmodel_t := update(worldmodel_{t-1}, input_t);
      commitment_t := deliberate(worldmodel_t);
      output_t := execute(commitment_t);
```

Figure 2: Stimulus-response architecture with Persistent World Model

As discussed above, efficiency and stability are reasons to memorize previous commitments to guide further decisions and actions. The deliberate-function can use complicated processes to evaluate possible future situations, it can make plans to guide the behavior for a longer time regarding coordination with other robots. It may be useful or even necessary (for stability) to consider the same commitment over several time steps. This means to have additional persistent states related to the future. This is considered by the architecture with persistent states for worldmodel and commitment as given in Figure 3. The essential difference to the stimulus response architectures is the treatment of commitment as a persistent mental state. It can be split further e.g. into desires, intentions, plans (BDI-architecture) with a lot of variants in the literature (cf. [Wooldridge99], [Burkhard00]). It serves for efficiency as well as for stability.

```
for t=0,1,2,...  do
      worldmodel_t := update(worldmodel_{t-1}, input_t);
      commitment_t := deliberate(commitment_{t-1}, worldmodel_t);
      output_t := execute(commitment_t);
```

Figure 3: Architecture with Persistent States for Worldmodel and Commitment

3.2 Layered Architectures

The concept of persistent states works fine as long as there is enough time for the calculations in a single interval between two time points t and $t+1$. But real time architectures in dynamical environments usually allow for fast action control (by `execute`) in short intervals, while sensor integration and update of the worldmodel as well as commitment need more time. There are severe synchronization problems.
A common used model is a hierarchical architecture where `execute` performs "low level" behavior with short time horizon and fast specification time, e.g. for collision avoidance. Each such low level behavior is realized by simple methods, e.g. predefined scripts or in the form of stimulus response behavior. The aim of the `deliberate`-function on the higher level is the choice of such a script, the computation of a plan etc. Following the necessities of "bounded rationality", the `deliberate` process constitutes a reduced "screen of admissibility" [Bratman87]:
If the environment is complex then longer computation time is necessary to analyze the global situation (e.g. for image processing and interpretation, modeling of other players, calculation of the utilities of different strategies etc.). But not all aspects of the global situation are subject to fast changes (e.g. the ball possession may change very rapidly, but the positions of players do not). Hence there is the possibility of shared work: Complex analysis is performed by the "global" `deliberate` calculations leading to search space reduction for "local" short time decisions of `execute`.
Classical layered two pass architectures have a control flow bottom up from lower layers to higher layers and then back again to the lowest layer. To act in time, the higher layers are used only if needed, or with lower frequency. In the first approach, the lower layers must decide if higher layers have to be involved. This can yield context problems as discussed in Section 4.2. In the second approach, higher layers have delays.
Layered one pass architectures have only one control flow through the layers. To act in time, the higher layers are called with lower frequencies. Implementations of one pass top-down architectures can use stack-oriented programming paradigms. Actions are pushed onto a stack. The action a from the top is executed if a is low level, otherwise it is replaced by lower level actions $a_1, \ldots, a_n$, respectively (cf. e.g. [dMARS]). Subroutine calls as used in (procedural) programming implement the same principle (using the run time stack). The computed commitment activates a subroutine which performs the necessary actions for the achievement of the committed goal. The control turns back to the higher level deliberation process for a new commitment when the subroutine has finished.
Such layered models have different time scales on their layers. This means that the synchronization between `deliberate` and `execute` is somewhat different to the description by Figure 3. The commitment can be understood as a (may be conditional) plan or script computed on the higher levels (by `deliberate`) at time t such that

$$\texttt{commitment}_t = \texttt{step}_t,\ \texttt{step}_{t+1}, \ldots,\ \texttt{step}_{t+k}\ .$$

The low level `execute`-function computes the outputs for $i = t, \ldots, t+k$ according to that script:

$$\texttt{output}_i := \texttt{execute}(\texttt{step}_i,\ \texttt{input}_i\ )\ .$$

The most recent input $\texttt{input}_i$ (or the worldmodel if available in time) is used for adaptation. A temporary stimulus response behavior is realized if identical steps $\texttt{step}_i$ are used. As an example we may think of the commitment to run to a certain position, where the `execute`-function has to realize the necessary movements over a longer time.
While `execute` is active at each time step t, the higher level commitments may remain unchanged over longer time intervals in the layered architectures. In fact, using the subroutine-paradigm, the higher level processes are inactive until the lower level processes are finished. A problem of these approaches is the difficulty of fast reactions on the higher levels to unexpected changes in the environment. The problem cannot be overcome by concurrent computations as far as the complete analysis of a global situation (including time consuming sensor processing and integration for the worldmodel, and future simulations, evaluations and means-ends-analysis for the commitment, respectively) consumes more time than only a single interval between two time points t and $t+1$. We will discuss this matter in Section 4, and the proposal of the "Double Pass Architecture" in Section 5 is an attempt to overcome the problem. The name "Double Pass Architecture" refers to a difference to the one pass and two pass architectures, such that two independent passes are performed top-down for the `deliberate`- and `execute`-functions, respectively.

4 Problems of Control

This section discusses some details of the problems concerning efficient controls. Efficiency means optimal behavior with respect to given constraints, especially complexity constraints ("bounded rationality").

4.1 Trade-offs

As discussed for layered architectures, worldmodel update and deliberation can be time consuming processes. An accurate analysis of the situation (i.e. by complex picture processing algorithms) is worthless if it comes too late. There is a **time-trade-off** between

- Precise decisions based on a sufficient analysis of the situation: It needs time for computing the perception from sensory data, its integration into the worldmodel, the calculation of possible outcomes of available activities, generation of appropriate plans etc.
  versus

- Immediate fast responses to the most recent sensory data: It leaves no time for complex deliberation.

Layered architectures distinguish between long term decisions (deliberations – which may need more time), and short term decisions (executions) guided by the long term ones. The guidance by the long term ones means a smaller scope of possible choices for the short term decisions and hence shorter computation times.
A problem of these architectures are delayed reconsiderations of long term plans: In the case of unexpected events, the adaptation of the long term commitments may come too

late. Therefore collision avoidance is usually part of the low level behavior. In football/soccer, the ball handling can be considered as low level behavior, too. But for the other players, their low level behavior (e.g. running) is not related directly to the ball. Nevertheless, they should react quickly e.g. in cases when a team mate looses the ball.
The **stability-trade-off** concerns the consideration and optimal handling of (unexpected) changes in the environment. It is a trade-off between

- Fast adaptation to new situations in order to act according to the most recent data, and according to the most promising alternatives, respectively,
  versus
- Stabile following of old plans in order to pursue an intention. Stabile behavior is important for resolved acting and for cooperation (to ensure trustiness).

Both alternatives have their drawbacks: Stability has the danger of fanaticism, i.e. pursuing of unachievable goals, like keep on running for a pass when the ball is already controlled by the opponents. Adaptation may lead to permanent changes of behavior like oscillations, e.g. changing directions while running to an object.
Adaptation may be very inefficient if the costs for adaptation itself are high over time (think of an undecided goalie which permanently revises the place of the ball for a goal kick). Therefore, the costs of adaptation have to be considered, too. The appreciation of adaptation costs and consequences are a matter of the time trade-off.
The both trade-offs are directly connected with persistent commitments: Time can be saved by memorizing commitments. The distinction between short and long term decisions helps to react faster. Stability needs the consideration of former decisions which can be memorized by persistent commitments. (But there exist other possibilities, e.g. the exploitation of physical properties like inertia.)
There are much more alternatives concerning the construction of robot controls. If the robot performs exact movements then the efforts of motion control can be reduced. Vice versa, inexact movements may be compensated by extensive motion control to some extend. This is a another trade off with consequences to deliberation and execution procedures.

4.2 Context

The context problem is best illustrated by the behavior of a player which does not control the ball. All he can do is changing his position using simple behaviors like run/walk/stay. Good positions are essential for the success of the team. The player has a lot of different alternatives related to many different goals. More than for the ball controlling agent, the optimal behavior depends on the global situation, i.e. of the context (like defensive or offensive play, distance to the ball / to other players / to the goals, actual score etc.).
In our first RoboCup implementations, position changing was handled on a global level. A unique procedure had to compute utilities for all situations. The calculation of useful utilities becomes more and more difficult with growing numbers of contexts. Thus, our results were rather raw.
A reasonable principle of agent architectures are hierarchical structures: Complex skills use simpler skills, complex options are structured by simpler options. Different positioning options can be embedded into larger options like goal defense, double pass etc. This makes deliberation easier: First the agent decides for a double pass, then he decides for the appropriate positioning actions in this context.
In the classical, sequential, stack-oriented software architectures this can be implemented by successively called subroutines: The "play soccer method" calls the "offensive play

method" which calls the "double pass method" which calls the "positioning method" at the right time. Only the most recently called method/procedure is active (here: "positioning method"), the callers wait for the termination of this method. Each subroutine in the stack can be considered as a "level" of hierarchically ordered commitments. The number of levels is not restricted. In contrast, classical layered architectures have a very limited number of layers (e.g. two or three) which implement different reasoning methods and which are called with different frequencies.

In the case of unexpected events, the successively called subroutines can react only on the lowest level, i.e. by the active subroutine. Classical layered architectures have related problems, they react only on the lowest layer. This is sufficient as long as the higher layers need no fast changes according to unexpected events. The long term goal of an unmanned ground vehicle usually does not change because of an unexpected obstacle. After drawing aside it will continue its way to the former goal. If there are still serious problems, it can stop for deliberation. Hence the concept of fast low level reactions works well for such scenarios.

But, if fast changes are needed on higher levels, then the considerations of unexpected events could be done best in the appropriate context. For example, the loss of the ball by the second player during a double pass should be handled by the "play soccer method". It should lead to termination of the "offensive play method" and its successively called methods ("double pass method", "positioning method"). At the same time, the "play soccer method" should activate the "defensive play method" with appropriate submethods, e.g. for attacking the opponents.

If only the "change position method" is active (e.g. as the active subroutine), all necessary computations (analysis of the situation, test of conditions, termination of higher level routines up to the "offensive play method") must performed by this method. This leads to a very complex and inefficient "change position method". Moreover, since the "change position method" is used in many other contexts, this method becomes overwhelming complex. Alternatively, different "change position method" could be implementing for different contexts. But this would lead to multiple copies of code.

The problem can be solved if the `execute` procedure has access to all levels, and if the decisions to be performed can be restricted. A solution is proposed in the following Section 5. The Double Pass architecture is intended to permit real time redeliberation on all levels using principles of bounded rationality.

5 The Double Pass Architecture

This section proposes an architecture which deals with the above problems in a reasonable way. It uses persistent mental states for the past (worldmodel) and for the future (commitment). It can implement goal-directed approaches, e.g. the BDI-approach [Bratman87]. It uses a hierarchical structure and a least commitment strategy. The hierarchical structure provides options on different levels. It is traversed by two independently running passes. The hierarchy allows to describe behaviors and plans in a unique way, ranging from single actions on the lowest level up to long term plans on the highest levels . The lower level behaviors are combined to higher level plans. The passes perform different tasks:

The Deliberator performs time consuming processes regarding all aspects of the recent situation like choice of goals and long term planning. It sets up a partial hierarchical plan. Following the least commitment idea, the plan is refined as time goes on. Normally the deliberator does not have time problems since he works with sufficient forerun. Time critical decisions are left to the executor.

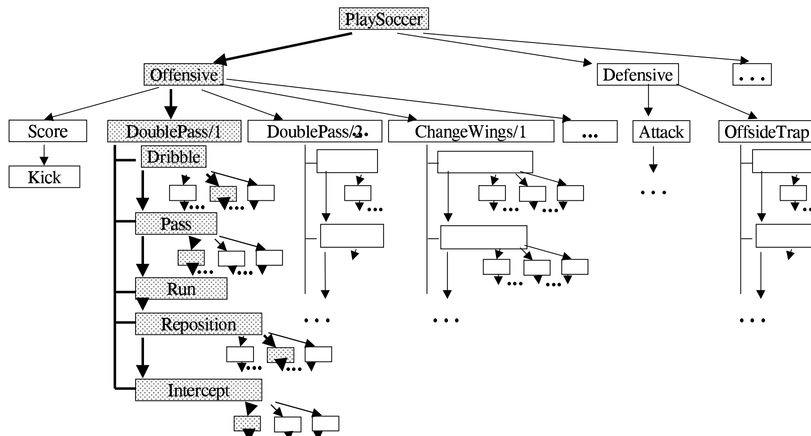


Figure 4: Option Tree with Intention Subtree

The Executor performs the contemporary decisions. Based on the preparatory work of the deliberator, its search space is restricted to a minimum of decisions using the most recent sensory information. In contrast to classical layered architectures, the executor considers all levels in real time.

Both lines of operation are independently running *passes* through all (!) levels of the hierarchy: Thus we have a *"Double Pass"* run time structure. This is in contrast to runtime organization in layered architectures (where short time decisions only affect the lowest level) and in programming languages (where only the procedure on the top of the stack is active).

5.1 Options

The data structure from which goals (or desires and intentions) are chosen from are the options. The set of options can be considered as a (virtual) tree structure with long term options near the root and specific short term actions near the leaves. An example from the football/soccer domain is given in Figure 4. The numbers (e.g. in DoublePass/1) denote the role (first player) in a cooperative behavior.
An option is performed by appropriate suboptions as defined by the tree. There are two kinds of connections between options and suboptions:

- Choice-Options can be performed by different, alternative suboptions (e.g. a pass can be performed by a forward-kick, a sideward-kick etc.), cf. Figure 5 for a Petri Net description of the alternatives of an offensive option. Transition firing depends on side conditions. "MaxUtility" means temporal priority for the transition with highest utility according to the recent situation. Other conditions are boolean valued.

- Sequencing-Options are performed by a sequence of suboptions (e.g. the suboptions of a double pass as described above), cf. Figure 6 for a Petri Net depicting the suboptions of the double pass option from the perspective of the player with the role DoublePass/1.

For clarity, the both kinds of connections are not mixed. This is similar to Prolog concepts: alternative suboptions correspond to different clauses of a predicate, sequenced suboptions correspond to the subgoals in a clause.

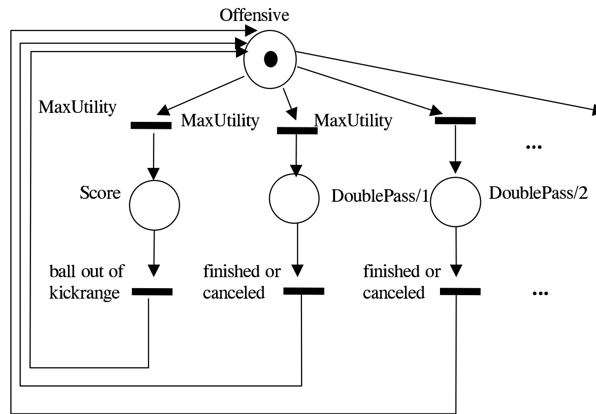


Figure 5: Example of a Choice-Option

Choice-options describe the different possibilities in the context of that option. Deliberator activities consist of choices from the alternative suboptions (e.g. using utilities), calculating appropriate parameters (e.g. the player to cooperate with in a double pass) and decisions concerning the termination (or cancellation) of intended activities. Alternative plans can be provided if a plan is canceled. The hierarchical structure allows for local decisions. Redeliberation (if needed) is performed in a given context.
Sequencing options describe the steps (suboptions) needed to perform a higher level option. There have to be well-defined criteria for the transitions from one suboption to the next one. The evaluation of these criteria is time critical because they are performed by the executor when acting in response to the newest sensory data.
According to deliberation and execution, options can be in different states. The deliberator chooses options/suboptions to be executed as *intentions/subintentions*, their state is then called "intended". They build a subtree of the option-tree as shown for a double pass in Figure 4. The complete intention subtree must contain one subintention for each choice-option starting in the root down to some leaves, and all subintentions for each sequencing option. Using the least commitment principle, the intention tree has the form of a hierarchical partial plan. Subintentions describe the plan parts on different levels.
At any concrete time point, there exists a unique path in the intention subtree (cf. Figure 4) from the root to a leave consisting of the *active* options. This path is called *activation path*. At the time when the first player passes to the second one, the activation path consists of "PlaySoccer"–"Offensive"–"DoublePass/1"–"Pass"– ... down to a concrete action (e.g. a kick-command with specified power and direction).
The executor performs the transition (as soon as the related condition is satisfied) from an active option to the subsequent option (as provided by the plan in the form of a sequencing option), and then the subsequent option becomes active. For example, after the pass is finished, the player starts running for a new position (cf. Figure 6). Transitions are checked (and performed if conditions are fulfilled) by the executor on all levels following the activation path.
Besides intentions, the deliberator can also prepare desires as candidates for forthcoming or alternative intentions. Desires build a subtree similar to intentions. The deliberator

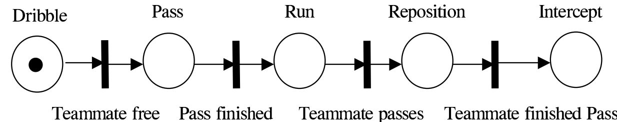


Figure 6: Example of a Sequencing Option

may choose between different desires when he has to decide for an intention. Desires can be used as fast available alternatives for the executor when he has to stop a plan according to unexpected situations. As an example we might think about the fast switch to scoring a goal (because the situation allows it) instead of continuing the double pass (a related transition can be added to the Petri Net in Figure 5).

5.2 Deliberator

The aim of the deliberator is the preparation of intentions as partial hierarchical plans (built from options) without any time stress (cf. Figure 4). It can prepare this plan (as a desire) while the executor is still performing an old intention. For example, the deliberator evaluates the available plans after an intercept while the robot is still running for the ball. At the same time, other players can evaluate their contributions to the possible plans of their team mate. As in real football/soccer, planning from stretch is difficult because of the indetermination of other player's behavior. Instead we can use so-called standard situations.
Standard situations provide generic cases of cooperative play. Using methods from Case Based Reasoning (CBR, cf. [Lenz-et-al98]), a concrete situation can be matched to the standard situation. For example, a triggering feature for the double pass is an opponent on the way of an offensive player controlling the ball. The standard situation (the "case") provides a standard scheme ("solution") for an intention. Using CBR methods for adaptation, a concrete intention can be specified. The option hierarchy serves as a structure for describing cases (cf. Section 6).
The deliberator computes long term decisions. It can be understood as the `deliberate`-function from Figure 3.

5.3 Executor

Short time behavior should rely on the newest available data: Hence there is no place for time consuming deliberations. The advances and the drawbacks of stimulus-response approaches and layered deliberative approaches have already been discussed. Stimulus-response architectures allow for fast reactions, but cannot handle complex long term behavior, while layered deliberative architectures can handle complex long term behavior, but have problems with dynamically changing situations.
The concept of the special executor pass through all layers is proposed as a solution. It works according to the recent activity path in the intention subtree. It starts from the root and proceeds level by level down to the leave which specifies the next output action to be executed by the robot. On each level it performs certain tests (e.g. if a subintention should terminate or stop), and it can calculate parameters according to the newest data (e.g. for performing an optimal kick). If a subintention is terminated, it performs the

transition to the next subintention. It may also switch to a desire and make it the new intention.
It is essential that all tests and calculations of the executor can be performed in short time, and that they are performed on the appropriate level. All time consuming computations should be performed by the deliberator in time before. The structure of options must be designed for these purposes.
The executor works as soon as new actions are to be performed, and as late as the newest data relevant for these actions can be analyzed. This can be done concurrently to the work of the deliberator - which at the same time prepares and specifies later activities for the executor. In a strictly sequential approach, the executor must interrupt the interpreter. Concrete implementations are possible in different ways, they are still in an experimental state.
The executor operates over the restricted search space of the intention tree provided by the deliberator. It can be understood as the implementation of the `execute`-function from Figure 3, but regarding all levels.

5.4 Main Features of the Double Pass Architecture

The option hierarchy allows for unique descriptions of behaviors and plans on different levels. All levels are treated the same way. An important feature of the Double Pass Architecture is the possibility of immediate reactions on all levels. It can be described as a "doubled one-pass-architecture": One-pass-architectures have a control flow which passes through each level only once. In our case, the control flow is directed top-down from the highest level to the lowest one. The difference consists in the fact, that there are two separated passes: One pass for the deliberator which prepares commitments (e.g. goal and plans), and another path for the executor which allows for real time reactions on all levels. The executor allows for a certain kind of stimulus-response behavior on all levels, where the stimulus-response behavior has been prepared by the deliberator. The executor realizes real-time behavior, while the deliberator acts without short time constraints.
Classical layered one- and two-pass architectures in complex dynamical environments have serious synchronization problems. Computations on higher (deliberative) layers are performed in longer time intervals, and rapid responses to changes in the environment are possible only at the lower (reactive) layers. The executor of the Double Pass Architecture works in short time intervals like the reactive components of classical layered architectures, but it passes through the higher levels, too. This is possible without synchronization problems since the deliberator prepares a restricted search space (the intention tree) for the executor.
The requirement to run through all levels by the executor needs a special runtime organization. Most runtime organization methods in programming are based on stacks, where a higher level method is called again only when the lower level has terminated. This holds for imperative languages as well as for descriptive ones, and it is used in agent architectures, too. The implementation of the new runtime strategy is still under work.

6 Conclusion, Further Work

The paper has discussed different aspects of basic approaches for robot/agent control. The notions of persistent states (concerning the past and the future, respectively) have been identified as characteristic concepts. Different approaches can be classified along these lines. They provide more clear differences than the classical notions of reactive and deliberative behavior. The classification helps to identify the problems of real time

controls in dynamical environments when long term planning is involved. The Double Pass Architecture was proposed to avoid the difficulties of layered architectures in dynamically changing environments. It allows for fast adaptations to new situations even on the higher levels.
Future plans include the use of the new architecture for robot learning. The project "Architectures and Learning on the Base of Mental Models" of the research program 1125 "Cooperating teams of mobile robots in dynamic and competitive environments" granted by the German Research Association (DFG) investigates the usage of CBR methods for control of robots. The cases correspond to generic behavior which can be specified and adapted according to the current situation. There are two main goals for using CBR: The first goal is the efficient control, while the second goal is learning from experience. Learning can be twofold: New cases can be acquired as templates for behavior, and the usage of existing cases can be improved by better analysis and adaptation methods.
There have already been some attempts to use CBR-methods for Robot Control, e.g. for opponent positioning models [Wendler/Lenz98], and for problems of self localization [Wendler et. al.00]. The investigations in opponent modeling have discovered a problem of dynamics when using CBR for low level behavior: As soon as the team tries to adapt to the opponents positions, the opponents did change to other positions. In the consequence, we need adaptation to higher level strategies. The option hierarchy serves as a structure for describing higher level cases. It gives room for off-line learning as well as on-line learning.

References

[Arkin98] Arkin, R.C.: Behavior Based Robotics. MIT Press,1998.

[Bratman87] M.E. Bratman. *Intentions, Plans, and Practical Reason.* Harvard University Press, Massachusetts, 1987.

[Brooks91] Brooks, R.A.: Intelligence without reason. Proceedings IJCAI-91, 569-595.

[Burkhard00] H.D. Burkhard: Software-Agenten. In: Günther Görz, Claus-Rainer Rollinger, Josef Schneeberger: Einführung in die Künstliche Intelligenz. Oldenbourg 2000, 941-1015. (In German)

[Burkhard02] H.D. Burkhard: Real Time Control for Autonomous Mobile Robots. To appear in Fundamenta Informaticae.

[Burkhard *et al.*98] Burkhard, H.D., Hannebauer, M. and Wendler, J.: Belief-Desire-Intention Deliberation in Artificial Soccer. *AI Magazine* 19(3): 87–93. 1998.

[dMARS] dMARS: Technical Overview - 25 JUN 1996.
www.aaii.oz.au/proj/dmars_tech_overview/dMARS-1.html

[Georgeff/Kinny97] Georgeff, M.P.; Kinny, D.N.: Modeling and Design of Multi-Agent Systems. In: M"uller, J.P.; Wooldridge, M.J.; Jennings, N.R. (Hrsg.): Intelligent Agents III, LNAI 1193, 1–20, Springer-Verlag, 1997

[Georgeff/Lansky87] M.P. Georgeff, A.L.Lansky: Reactive reasoning and planning. Proc. AAAI-87, 677–682, 1987.

[Kitano-et-al-97] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara. RoboCup: A challenge problem for ai. *AI Magazine*, 18(1):73–85, 1997.

[Lenz-et-al98] Lenz, M., Bartsch-Spörl, B., Burkhard, H. D., Wess, S. (Eds.): Case Based Reasoning Technology. From Foundations to Applications. LNAI 1400, Springer 1998.

[Maes90] Maes, P. (Hrsg.): Designing Autonomous Agents. Theory and Practice from Biology to Engineering and Back. MIT Press 1990

[JPMüller96] M"uller, J.P.: The Design of Autonomous Agents – A Layered Approach. LNAI 1177, 1996.

[Müller-Gugenberger/Wendler98] Müller-Gugenberger, P. and Wendler, J.: *AT Humboldt 98 — Design, Implementierung und Evaluierung eines Multiagentensystems für den RoboCup-98 mittels einer BDI-Architektur*. Diploma Thesis. Humboldt University Berlin, 1998.

[Murphy00] Robin R. Murphy: Introduction to AI. MIT Press, 2000.

[Parunak97] Van Parunak: 'Go to the Ant': Engineering Principles from Natural Agent Systems. Annals of Operations Research, 1997.

[Rao/Georgeff91] A. S. Rao and M. P. Georgeff. Modeling agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proc. of the 2rd Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'91)*, 1991.

[RoboCup] RoboCup. The Robot World Cup Initiative: `www.robocup.org`
Annual Proceedings of the RoboCup Workshops/Symposia appear in the Springer LNAI-Series.

[Russell/Norvig95] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice-Hall, 1995.

[Weiß99] Weiß, G. (Hrsg.): Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence, MIT Press 1999.

[Wendler et. al.00] J.Wendler, S.Brüggert, H.D.Burkhard, H.Myritz: Fault-tolerant Self Location by Case Based Reasoning. In P.Stone, T.Balch, G.Kraetzschmar (eds.): RoboCup 2000: Robot Soccer World Cup IV. Springer LNAI 2019, 259-268.

[Wendler/Lenz98] J.Wendler and M.Lenz: CBR for Dynamic Situation Assessment in an Agent-Oriented Setting. In D.Aha and J.Daniels (eds.): Proc. of the AAAI-98 Workshop on Case-Based Reasoning Integrations, 1998, Madison, USA.

[Wooldridge99] Wooldridge, M.: Intelligent Agents. In [Weiß99], pp. 27–78.

Formal Model of Joint Achievement Intention *

Mao Xinjun[1] Wu Gang[1] Wang Huaimin[1] Zhao Jianming[2]

[1] (National Lab. for Parallel and Distributed Processing, China)

[2] (School of Computer Science, Zhejiang Normal University, China)

E-mail: xjmao21@21cn.com

Abstract: The joint achievement intention represents the common task of agents to achieve collectively and is an important concept to specify and analyze the social behaviors in multi-agent system. The paper discusses the meaning and characteristics of joint achievement intention, analyzes the limitation and problems in existing work, defines the joint achievement intention with new and clear semantics based on the logic framework of multi-agent system, specifies and proves its important properties. The novel formal model of joint achievement intention can be used to effectively support the development of multi-agent system.

Key words: multi-agent system, joint achievement intention, belief

1. Introduction

Agent is an encapsulated computational entity that is situated in some environment, and that is capable of flexible, autonomous action in that environment in order to meet design objectives [11]. Multi-agent system is composed of a number of interacting and cooperating agents, each of them having limited capabilities and resources. As the abstract model that agent techniques provide can express the computational entities and problem-solving manner in applications more naturally and effectively, much attention has been imposed on the researches of agent techniques nowadays.

In multi-agent system, as the dependencies among agents' actions, the limitation of each agent's capabilities, and the distribution of system's resources, the joint work among agents is absolutely necessary to meet global constraints and natural problem solving. The joint work among agents represents the social behaviors in multi-agent system. In order to develop the multi-agent system, we must put forward some effective tools to specify and analyze such social behaviors, for instance, what is the social behaviors among agents, how will it affect agent's actions, how is it related with agent's internal state such as belief and intention, etc.

In the area of artificial intelligence, agent is taken as an intentional system with such cognitive components as belief, goal, intention, etc. The representative work is the BDI agent architecture. However, the agent's internal cognitive components only define individual behaviors and, as such, are an insufficiently rich base on which to build a principled representation of social behaviors. There are two main limitations with the individualistic approach. Firstly, joint action is more than just the sum of individual action, even if the action happens to be coordinated. For example, it will be somewhat unrealistic to claim that there is

* The paper is supported by Natural Science Foundation of China with Granted No: 60003002 and 60103009, and National 973 Project G1999032700, and ZheJiang Natural Science Foundation with Granted No ZD0108.

any real teamwork involved in ordinary automobile traffic, even though the drivers act simultaneously and are coordinated by traffic signs. Secondly, there is a fundamental difference between individuals and groups [10]. Therefore, some new abstract concept model is needed to describe and investigate the joint social behaviors in multi-agent system.

Joint achievement intention is an important abstract concept in distributed artificial intelligence to examine the social behaviors in multi-agent system. The paper is structured as follows. The paper discusses the meaning and characteristics of joint achievement intention informally (see section 2), introduces the existing research work on joint achievement intention and analyzes their limitation (see section 3), and based on logical framework of multi-agent system (see section 4), defines the formal and rigorous semantics of joint achievement intention, specifies and proves its properties (see section 5). The final section concludes the paper and outlines directions for future work.

2. Characteristics of Joint Achievement Intention

The joint achievement intention of agent means that agents will together achieve some preposition by joint behaviors and represents the common task of agents to achieve. It corresponds to the joint intention concept in [2,3,4,11]. In multi-agent system, the purposes of joint social behaviors among agents are not only to achieve some collective tasks, but also to maintain the system state. For instance, there are two robot agents in some environment, which are assigned the task to move objects from one place to another place cooperatively. In order to meet the system constraints, some restriction that the object must be moved horizontally and stably are imposed on the joint social behaviors of the two agents. Obviously, such a restriction will affect the action choice of the related agent in the process of the coordination. However, the traditional meaning and theory of joint intention concept cannot represent and analyze such system constraints, which widely exist in multi-agent system. Therefore, we have introduced a new and novel concept of joint maintenance intention (the work is introduced in other paper). In order to distinguish the joint intention from joint maintenance intention, we give a new name to joint intention as joint achievement intention.

Intuitively, the joint achievement intention has the following properties.

- Action choice, which exhibits some rational choice for future joint behaviors and will restrict the agents' actions. The joint achievement intention is the factor that promotes agents to take joint social behaviors;
- Relativity, including mutual belief and cooperation during the process of joint social behaviors;
- Satisfiable, which means that the joint achievement intention of agents is achievable;
- Persistent, which means agents will not abandon their joint achievement intention in the process of joint social behaviors, and exhibits some commitment to joint social behaviors;
- Consistent, which means that it is consistent among several joint achievement intentions, and agents' joint achievement intention should be consistent with individual agent's internal state;
- Non-conflict, which means that there is no conflict among several joint achievement

intentions of agent;

- Consistent with belief, which means that agents' joint achievement intention should be consistent with agents' belief. If agents have some joint achievement intention, then they should believe that the joint achievement intention should be achievable.

3. Evaluation of Existing Work

There are many researchers in the area of computer science and philosophy investigating joint intention. The representative research is Cohen and Levesque's work [2], which introduces joint intention concept to handle cooperative intentions. Joint intentions are intended to clarify the relationships among beliefs, desires and intentions for multiple agents.

Joint intentions' theory of Cohen and Levesque is developed in three levels. Firstly, they define *weak goals*, which specify the conditions under which an agent holds a goal, and the actions it must take if the goal is satisfied or impossible.

$$\boldsymbol{WG}(x, y, p) = (\neg Bel(x,p) \wedge Goal(x, \Diamond p)) \vee (Bel(x,p) \wedge Goal(x, \Diamond MB(x, y, p))) \\ \vee (Bel(x, \Box\neg p) \wedge Goal(x, \Diamond MB(x,y, \Box\neg p)))$$

The above definition means that $\boldsymbol{WG}(x, y, p)$ is satisfied if and only if one of the following conditions hold: (1) agent x believes that p is not true and desires p to be true at some future time; (2) agent x believes that p is already true and desires that y also mutually believes that p is true; (3) agent x believes that p will never be satisfied and wants y to mutually believes that p will never be satisfied.

Secondly, they define *joint persistent goals* for multiple agents.

$$\boldsymbol{JPG}(x,y,p,q) = MB(x, y, \neg p) \wedge MG(x, y, p) \wedge \\ Until(MB(x,y,p) \vee MB(x,y, \Box\neg p) \vee MB(x,y, \neg q) , MB(x,y,(MG(x,y,p) \wedge MG(y,x,p))))$$

In order to hold a joint persistent goal, agents must therefore: (1) Mutually believe that the p is not satisfied; (2) Hold p as a mutual goal; (3) Hold p as a weak mutual goal until either they mutually believe that p is satisfied, or they mutually believe that p will never be satisfied, or they mutually believe that some other condition q will never be satisfied.

Finally, they define *joint intentions* in terms of weak goals and joint persistent goals.

$$\boldsymbol{JI}(x,y,a,q) = JPG(x,y,DONE(x,y, Until(DONE(x,y,a), MB(x,y,DOING(x,y,a)))?;a),q)$$

The theory of Cohen and Levesque gives entire meaning of joint intention. However, there are a number of limitations and shortcomings in it. Firstly, action choice is the basic and essential characteristics of joint intention. Their work defines the semantics of joint intention concept based on the linear temporal logic and possible world model. Therefore, the semantics definition of joint intention cannot well capture the action choice characteristics. Secondly, the semantics definition of joint intention includes the modification strategy of joint intention, which not only cannot well describe the essential meaning of joint intention, but also make the theory much more complicated. Thirdly, the semantics definition is based on possible world model, which has logical omniscience problem.

Other researches include Nunes, Raimo, Jennings's work [3,4,10]. Some of them extend Cohen and Levesque's theory, others investigate the joint intention from the point of philosophy. Here we will not introduce and give comments on them.

4. Logical Framework

The semantics definition of joint intention is based on logical framework of multi-agent system, which includes three parts: syntax, model and semantics.The formal language L is the extension of branch temporal logic *CTL**[7], which is composed of two parts: state formulas $\boldsymbol{L_t}$ and path formulas $\boldsymbol{L_s}$ defined in the follows. Let Φ is the atomic proposition symbol set, $Const_{ag}$ agent symbol set. To simplify description, the paper has the following symbol convention: p, q,⋯ as proposition symbol, and φ, ψ, … as formula, and x, y, … as agent symbol.

Definition 4.1 (Syntax of Language *L*) The formal language L is the smallest closed set defined by the following rules

(1) if $p \in \Phi$, then $p \in \boldsymbol{L_t}$

(2) if $\psi, \varphi \in \boldsymbol{L_t}$ and $x, y \in Const_{ag}$, then $\neg\varphi$, $\psi\wedge\varphi$, $\boldsymbol{Bel}(x, \varphi)$, $\boldsymbol{MB}(x, y, \varphi)$, $\boldsymbol{AI}(x, y, \varphi)$, $\boldsymbol{MAI}(x, y, \varphi)$, $\boldsymbol{MAB}(x, y, \varphi)$, $\boldsymbol{WAC}(x, y, \varphi)$, $\boldsymbol{MAC}(x, y, \varphi)$, $\boldsymbol{JAI}(x, y, \varphi) \in \boldsymbol{L_t}$

(3) $\boldsymbol{L_t} \subseteq \boldsymbol{L_s}$

(4) if $\psi, \varphi \in \boldsymbol{L_s}$, $x \in Const_{ag}$, then $\neg\varphi$, $\psi\wedge\varphi$, ψ ***Until*** φ, $\psi \boldsymbol{Until}_{\forall}\, \varphi \in \boldsymbol{L_s}$

(5) if $\varphi \in \boldsymbol{L_s}$, then $\boldsymbol{A}\varphi \in \boldsymbol{L_t}$

Definition 4.2 (Formal Model of *L*) The formal model of $\boldsymbol{L}$ is defined as $\boldsymbol{M} = < T, <, U_{ag}, \pi, [\,], B, C >$, where T is moment set, each member of which representing a world state. < is a partial order on T, which describes the temporal order among moments. The past of each moment is deterministic and linear. It's future may be branching. Figure1 give a schematic description of formal model that is tree-like structure. U_{ag} is an agent set. $\pi: \Phi \rightarrow \wp(T)$ defines the moment set at which p is satisfied. $[\,]$ defines the assignment to agent symbol. $B: U_{ag} \rightarrow \wp(T\times T)$, $(t, t') \in B(x)$ means that at moment t agent believes that moment t' is possible and is used to define agent's belief.

A path at moment t describing some way that the world may evolve is a branch that evolves from t and is composed of future moment of t.

Definition 4.3 (Path) A path of moment t is a set $S \subseteq T$ which satisfies：(1) $t\in S$；(2) $\forall t_1, t_2 \in S$：$(t_1<t_2) \vee (t_2<t_1) \vee (t_1= t_2)$；(3) $\forall t_1, t_2\in S; t_3\in T$：$(t_1<t_3<t_2) \Rightarrow (t_3\in S)$；(4) $\forall t_1 \in S; t_2\in T$：$(t_1< t_2) \Rightarrow (\exists t_3\in S$：$(t_1<t_3) \wedge \neg (t_3 < t_2))$；(5) $\forall t_1 \in S : (t = t_1) \vee (t < t_1)$

(1) denotes that the path of moment t contains t; (2) describes the linearity property of the path; (3) describes the density property; (4) describes the relative maximum; (5) denotes the initiate of the path . Let $\boldsymbol{S_t}$ the set of all paths at moment t, $\boldsymbol{S_\Sigma}$ the set of all paths.

In multi-agent system, each agent takes actions concurrently and asynchronously. At any moment, agent can take action to influence and control the way that the world evolves. However, such an influence is limited, because the way that the world may evolve is also influenced and controlled by the environment events and the actions that other agents take. All of the actions taken by agents in multi-agent system and the environment events together determine the evolution way of the world.

For example, Figure1 describes a formal model of a multi-agent system composed of two agents. The node in the figure denotes moment represented by a set of propositions. The edge denotes the combination of actions taken by all agents. The symbol "||" denotes that the actions of several agents are taken concurrently. In order to simplify description, we assume that the

action symbol on the left part of "||" represents the action of $agent_1$ and the right part represents the action of $agent_2$. At moment t_0 $agent_1$ can take action *a* to make the world evolve to the moment t_1 or t_2, or take action *b* to make the world evolve to the moment t_3 or t_4. But when $agent_1$ take action *a*, whether the world evolves toward moment t_1 or moment t_2 is also depended on the action taken by $agent_2$. When $agent_2$ takes action *c*, then the world evolves to moment t_1. If $agent_2$ takes action *d*, then the world evolves to moment t_2.

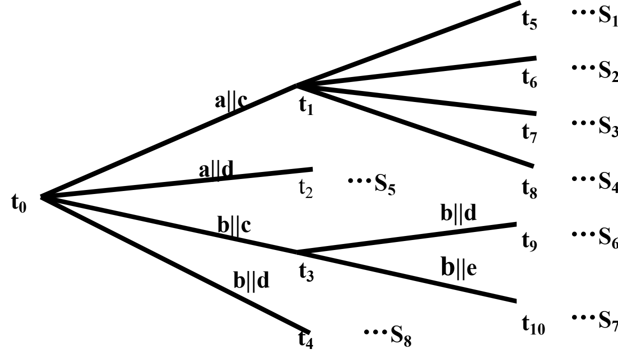


Figure1. formal model of multi-agent system

C： $U_{ag}\times T\rightarrow \wp(\mathbf{S}_\Sigma)$ is to define joint intention of agent, where C(x, t) is the path set that agent chooses at moment *t* and $C(x,t)\subseteq \mathbf{S}_t$. To simplify research, there is a model constraint: $\forall t\in T, x\in U_{ag}: C(x, t) \neq \varnothing$.

The satisfactory semantics of the formula in $\boldsymbol{L_t}$ is defined by model M and moment *t*. $M \models_t \varphi$ denotes that formula φ is satisfied at the moment *t* in model M. The satisfactory semantics of the formula in $\boldsymbol{L_s}$ is defined by model M, path S and moment t. $M \models_{s,t} \psi$ denotes that formula ψ is satisfied at moment *t* on path *S* of moment M.

Definition 4.4 (Semantics of *L*)

(1) $M \models_t p$ iff $t\in\pi(p)$

(2) $M \models_t \psi \wedge \varphi$ iff $M \models_t \psi$ and $M \models_t \varphi$

(3) $M \models_t \neg\varphi$ iff $M \not\models_t \varphi$

(4) $M \models_t \boldsymbol{A}\varphi$ iff $\forall S: S\in \mathbf{S_t} \Rightarrow M\models_{s,t} \varphi$

(5) $M \models_t \boldsymbol{Bel}(x, \varphi)$ iff $\forall t': (t, t')\in B(\lbrack x)) \Rightarrow M \models_{t'} \varphi$

(6) $M \models_t \boldsymbol{MB}(x, y, \varphi)$ iff $M \models_t \boldsymbol{Bel}(x, \varphi)$ and $M \models_t \boldsymbol{Bel}(y, \varphi)$

(7) $M \models_{s,t} \psi \wedge \varphi$ iff $M \models_{s,t} \psi$ 且 $M \models_{s,t} \varphi$

(8) $M \models_{s,t} \neg\varphi$ iff $M \not\models_{s,t} \varphi$

(9) $M \models_{s,t} \psi \boldsymbol{Until} \varphi$ iff $\exists t'\in S: (t \leq t')$ and $(M \models_{s,t'} \varphi)$ and $(\forall t'': t \leq t'' < t' \Rightarrow M \models_{s,t''} \psi)$

(10) $M\models_{s,t} \psi \boldsymbol{Until}_\forall \varphi$ iff $\forall t'\in S: (\forall t'': t \leq t'' \leq t' \Rightarrow M \models_{s,t''} \neg\varphi) \Rightarrow M \models_{s,t'} \psi$

(11) $M \models_{s,t} \varphi$ iff $M \models_t \varphi$, where $\varphi\in \boldsymbol{L_t}$.

Other operators can be derived based on the above semantics definition. ***Until*** is a "until" operator, $\boldsymbol{Until}_\forall$ is a weak "until" operator. $\boldsymbol{F}\varphi$ = true ***Until*** φ is existential temporal operator. ***G*** is the dual of ***F*** and is universal temporal operator. ***A*** is universal path operator, $\boldsymbol{A}\varphi$ is satisfied at moment *t* if and only if φ is satisfied on all paths of moment *t*. ***E*** is the dual of ***A***, that is $\boldsymbol{E}\varphi = \neg\boldsymbol{A}(\neg\varphi)$, therefore ***E*** is existential path operator. $\boldsymbol{Bel}(x, \varphi)$ denotes that $agent_x$ has belief φ. Here, we assume that B(x) is reflexive and transitive. Therefore ***Bel*** corresponds to the modal operator in S4 normal modal system.

Theorem 4.1 ***Bel*** has the following properties:

(1) $|= \boldsymbol{Bel}(x, \varphi) \rightarrow \varphi$

(2) $|= \boldsymbol{Bel}(x, \varphi) \rightarrow \boldsymbol{Bel}(x, \boldsymbol{Bel}(x,\varphi))$

(3) $|= \boldsymbol{Bel}(x, \varphi) \wedge \boldsymbol{Bel}(x, \varphi\rightarrow\psi) \rightarrow \boldsymbol{Bel}(x, \psi)$

(4) if $|= \varphi$, then $|= \boldsymbol{Bel}(x, \varphi)$

5. Joint Achievement Intention Theory

We will extend the intention theory in [1] to establish the theory framework of joint achievement intention in multi-agent system, investigate the relationship between joint achievement intention with the belief and intention of individual agent.

5.1. Intention

The achievement intention of individual agent means that agent intends to achieve some preposition, which is the abstract representation of agent's task and goal. The formal model of multi-agent system is a tree-like structure, each branch at any moment denoting the possible choice that agent may select. The achievement intention φ of agent means that agent selects the world evolving paths, on each of which agent believes that φ will be satisfied eventually.

Definition 5.1.1 (Achievement Intention) $M|=_t \boldsymbol{AI}(x, \varphi)$ iff

$M|=_t \boldsymbol{Bel}(x, \neg\varphi)$ and $(\forall S: S\in C(x, t) \Rightarrow M |=_{s,t} \boldsymbol{FBel}(x, \varphi))$

The above semantics defines the basic characteristic of achievement intention, i.e., action choice. Different from existing methods, we define achievement intention as the choice of the world evolving paths, not the accessible relationship between possible worlds. In the formal model, the world evolving path is related to agent's action. Such a definition not only clearly describes the choice characteristic of achievement intention and how agent's intention will restrict agent's future behaviors.

5.2. Joint Achievement Intention

Action choice and relativity are the basic and essential characteristics of joint intention. In the following we will define the semantics of joint intention based on the two characteristics, analyze and validate its important properties. One of the preconditions of joint intention is that the both sides of agents have the common choice, viz. with the same achievement intention.

Definition 5.2.1 (Common Achievement Choice) $\boldsymbol{MAI}(x, y, \varphi) = \boldsymbol{AI}(x, \varphi) \wedge \boldsymbol{AI}(y, \varphi)$

Two agents having the common choice do not mean that they have the joint intention. The relativity condition must be satisfied before forming joint intention, which includes two parts: mutual belief intention and cooperation.

Definition 5.2.2 (Mutual Achievement Belief) $\boldsymbol{MAB}(x, y, \varphi)= \boldsymbol{MB}(x, y, \boldsymbol{AI}(x, \varphi)\wedge\boldsymbol{AI}(y, \varphi))$

$\boldsymbol{MAB}(x, y, \varphi)$ means $agent_x$ and $agent_y$ mutual know that the two sides have the achievement intention φ. In order to define the semantics of mutual achievement cooperation, we firstly introduce the "weak achievement cooperation" concept.

Definition 5.2.3 (Weak Achievement Cooperation) $\boldsymbol{WAC}(x, y, \varphi) =$

$(\boldsymbol{Bel}(x, \varphi) \wedge \neg \boldsymbol{Bel}(x, \boldsymbol{Bel}(y, \varphi)) \rightarrow \boldsymbol{AI}(x, \boldsymbol{MB}(x, y, \varphi))) \quad \wedge$

$(\boldsymbol{Bel}(x, \boldsymbol{AG}\neg\varphi) \wedge \neg \boldsymbol{Bel}(x, \boldsymbol{Bel}(y, \boldsymbol{AG}\neg\varphi)) \rightarrow \boldsymbol{AI}(x, \boldsymbol{MB}(x, y, \boldsymbol{AG}\neg\varphi)))$

AG¬φ means that φ will never be satisfied on all paths. ***WAC***(x, y, φ) denotes that $agent_x$ has weak achievement cooperation with $agent_y$ with regard to φ. ***WAC***(x, y, φ) is satisfied, if and only if, when $agent_x$ knows that φ is satisfied and does not know that $agent_y$ knows φ is satisfied, then $agent_x$ intend to let two sides know φ is satisfied, and when $agent_x$ knows that φ will never be satisfied and does not know that $agent_y$ knows φ will never be satisfied, then $agent_x$ intend to let two sides know φ will never be satisfied. In term of the weak achievement cooperation, the semantics of mutual achievement cooperation is defined as follows.

Definition 5.2.4 (Mutual Achievement Cooperation) $M \models_t \boldsymbol{MAC}(x, y, \varphi)$ iff

$(\forall S: S \in C(x,t) \Rightarrow M \models_{s,t} (\boldsymbol{MB}(x, y, \boldsymbol{WAC}(x, y, \varphi) \wedge \boldsymbol{WAC}(y, x, \varphi)))\ \boldsymbol{Until}\ \neg \boldsymbol{AI}(x, \varphi))$

and $(\forall S: S \in C(y,t) \Rightarrow M \models_{s,t} (\boldsymbol{MB}(x, y, \boldsymbol{WAC}(x, y, \varphi) \wedge \boldsymbol{WAC}(y, x, \varphi)))\ \boldsymbol{Until}\ \neg \boldsymbol{AI}(y, \varphi))$

MAC(x, y, φ) denotes that $agent_x$ and $agent_y$ have the mutual achievement cooperation φ. ***MAC***(x, y, φ) is satisfied if and only if the both sides mutual know that they will cooperate with each other until they drop their achievement intention.

Definition 5.2.5 (Joint Achievement Intention)

$\boldsymbol{JAI}(x, y, \varphi) = \boldsymbol{MAI}(x, y, \varphi) \wedge \boldsymbol{MAB}(x, y, \varphi) \wedge \boldsymbol{MAC}(x, y, \varphi)$

JAI(x, y, φ) denotes that $agent_x$ and $agent_y$ have the joint achievement intention φ. ***JAI***(x, y, φ) is satisfied, if and only if, that they have the common choice, and they know their common choice, and they know they will cooperation with each other during the process of achieving joint intention. The above semantics definition clearly and exactly describes the essential characteristic of joint achievement intention, viz. action choice and relativity. Based on the semantics, a number of important properties of joint achievement intention can be specified and proved.

Theorem 5.2.1 $\models \boldsymbol{JAI}(x, y, \varphi) \rightarrow \boldsymbol{MB}(x, y, \neg\varphi)$

The theorem describes the condition under which agents will accept or drop their joint achievement intention. $agent_x$ and $agent_y$ have joint achievement intention φ only if $agent_x$ and $agent_y$ mutual believe that φ is not satisfied. Rational agents will not jointly achieve proposition that is already satisfied. The theorem can be proved by the semantics definition of ***JAI***, ***MB*** and ***AI.***

Theorem 5.2.2 (Consistent) $\models \neg(\boldsymbol{JAI}(x, y, \varphi) \wedge \boldsymbol{JAI}(x, y, \neg\varphi))$

The theorem denotes that the joint achievement intentions of agents should be consistent. At any moment, agents cannot have the joint achievement intention φ and at the same time have the joint achievement intention ¬φ. The theorem can be proved by theorem 5.2.1 and theorem 4.1.

Theorem 5.2.3 (Consistent with individual's achievement intention)

$\models \neg (\boldsymbol{JAI}(x, y, \varphi) \wedge (\boldsymbol{AI}(x, \neg\varphi) \vee \boldsymbol{AI}(y, \neg\varphi)))$

The above theorem shows that agent's joint achievement intention is consistent with individual agent's internal achievement intention. At any moment, agent will not have the joint achievement intention φ and at the same time have the achievement intention ¬φ. According to the theorem 4.2.1, together with the semantics of ***JAI***、***MB*** and ***AI***, the theorem can be proved.

Theorem 5.2.4 (Satisfiable) $\models \boldsymbol{JAI}(x, y, \varphi) \rightarrow \boldsymbol{MB}(x, y, \boldsymbol{EF}\varphi)$

EFφ denotes that there exists a path on which φ will be eventually satisfied. Agents' joint

achievement intention should be satisfiable, or achievable. If $agent_x$ and $agent_y$ have the joint achievement intention φ, then they should mutual believe that φ is satisfiable. The theorem can be proved according to semantics of ***JAI***, ***MAB***, ***AI***, and the model constraint: $\forall t \in T, x \in U_{ag}$: $C(x, t) \neq \emptyset$ in section 4.

Theorem 5.2.5 (Consistency with belief)

$\models \neg\, (\boldsymbol{JAI}(x, y, \varphi) \wedge (\boldsymbol{Bel}(x, \neg \boldsymbol{EF}\varphi) \vee \boldsymbol{Bel}(y, \neg \boldsymbol{EF}\varphi)))$

The theorem shows that agents' joint achievement intention should be consistent with agents' belief. Agent will not have the joint achievement intention φ and at the same time believes that φ is not achievable. The theorem can be proved by theorem 5.2.4 and theorem 4.1.

Theorem 5.2.6 (Non-Conflict) $\models \boldsymbol{JAI}(x, y, \varphi) \wedge \boldsymbol{JAI}(x, y, \psi) \rightarrow \boldsymbol{MB}(x, y, \boldsymbol{E}(\boldsymbol{F}\varphi \wedge \boldsymbol{F}\psi))$

The theorem shows that the joint achievement intentions of agent should be non-conflict. If $agent_x$ and $agent_y$ have joint achievement intention φ and ψ, then they mutually believe that there exists a path on which φ and ψ will be eventually satisfied respectively. The theorem can be proved by semantics of ***JAI***, ***MAB***, ***AI***, and theorem 4.1.

Persistency is another important property of joint achievement intention. The joint achievement intention has the following persistency axiom.

Axiom 5.2.1 (Persistency Axiom)

$\boldsymbol{A}(\boldsymbol{JAI}(x, y, \varphi) \rightarrow \boldsymbol{JAI}(x, y, \varphi)\ \boldsymbol{Until}_{\forall}\ (\boldsymbol{MB}(x, y, \varphi) \vee \boldsymbol{MB}(x, y, \neg \boldsymbol{EF}\varphi)))$

The above axiom shows that if $agent_x$ and $agent_y$ have the joint achievement intention φ, then agent will persistently hold the joint achievement intention until they know that φ is satisfied or is impossible to achieve. In order to make the axiom sound, some constraint is imposed on the formal model: $\forall t \in T; S \in \mathbf{S}_{\Sigma}; x, y \in U_{ag}: M \models_t \boldsymbol{JAI}(x, y, \varphi) \Rightarrow$

$(\forall t' \in S: (\forall t'': t \leq t'' \leq t' \Rightarrow M \models_{t''} \neg(\boldsymbol{MB}(x, y, \varphi) \vee \boldsymbol{MB}(x, y, \neg \boldsymbol{EF}\varphi))) \Rightarrow M \models_{t'} \boldsymbol{JAI}(x, y, \varphi))$

6. Conclusion

The paper explains the significance and importance of making research on the joint achievement intention, analyzes the existing work and its limitations, discusses the characteristics of joint achievement intention. Based on the logical framework of multi-agent system, the new semantics of joint achievement intention is defined, a number of important properties are specified and proved. Different from existing work, the semantics definition of joint achievement intention is not based on the possible world accessible relation, but on the choice of world evolving paths in the formal model. We don't incorporate the modification strategy of joint achievement intention into its semantics definition. Such a semantics definition clearly captures and describes the basic and essential characteristics of joint achievement intention.

The theorem framework of joint achievement intention can effectively support analysis and design of multi-agent system, especially investigation of social behaviors of multi-agent system. The further work includes to specify and validate the interaction behaviors and cooperation model based on the theorem framework.

Reference

1. P.R.Chen, H.J.Levesque.Intention is choice with commitment. *Artificial Intelligence*, 1990, 42(2-3) : 213~261.
2. P.R.Chen and H.J.Levesque. Teamwork. Nous, 21, 1991.
3. Raimo Tuomela. Collective and joint intention. *Proceeding of cognitive theory of social action*, 1998.
4. Raimo Tuomela. Philosophy and distributed artificial intelligence: The case of joint intention. *Foundation of distributed artificial intelligence*, John Wiley&Sons, 1996: 487~504.
5. Mao Xinjun, Wang Huaiming. The intention theory of agent computing in multi-agent system, *Journal of software*, 1999, 10(1): 43~48.
6. Singh M P. Multiagent System: A Theoretical framework for Intentions, Know-how, and Communications. Berlin, Heidelberg: Springer-Verlag, 1994.
7. B.Chellas. Modal logic: an introduction. Cambridge University Press, Cambridge, MA,1980.
8. B. Dunin-Keplicz and R.Verbrugge. Collective Commitment. In *Proceeding of the second international conference on multi-agent system*, Menlo Park,CA,1996.
9. Panzarasa, P and N.R.Jennings. Social mental shaping: Modelling the impact of sociality on the mental state of autonomous agents. *Computational Intelligence*, 2001.
10. N.R.Jennings. Specification and implementation of belief-desire-joint intention architecture for collaborative problem solving. *Journal of Intelligent and Cooperative Information Systems*, 1993, 2 (3): 289~318.
11. N.R.Jennings. Building complex software system: the case for an agent-based approach. *Communication of ACM*, 2001.

Generic Multi-Agent Architectures for Multimedia Multimodal Dialogs

H. DJENIDI[1], A. RAMDANE-CHERIF[2], Pr. C. TADJ[1] and Pr. N. LEVY[2]

[1]*Electrical Engineering Department, École de Technologie Supérieure*
1100, Notre-Dame Ouest, H3C 1K3, Montreal, Quebec, Canada.
{hdjenidi,ctadj}@ele.etsmtl.ca

[2]*PRiSM, Université de Versailles St.-Quentin,*
45, Avenue des Etats-Unis, 78035 Versailles Cedex, France
{rca, nlevy}@prism.uvsq.fr

Abstract: The multimodal featuring fusion for natural human-computer interaction involves complex intelligent architectures facing unexpected errors and mistakes made by users. These architectures should react to events that occur simultaneously with eventual redundancy from different input media. In this paper, intelligent agent based generic architectures for multimedia multimodal dialog protocols are proposed. Global agents are decomposed into their relevant components. Each element is modeled separately using timed Colored Petri networks. The elementary models are then linked together to obtain the full architecture. Hence, maintainability, understandability and the modification of the architecture are facilitated. For validation purpose, the proposed multi-agent architectures are applied on a practical example.

Keywords: Multimedia multimodal dialog fusion, Multi-agent architecture, Timed Colored Petri networks.

1 Introduction

With the growing technology, many applications supporting more transparent and flexible human computer interactions have emerged. This results in an increasing need for more powerful communication protocols, especially when several media are involved. Multimedia multimodal applications mean systems combining natural input modes, such as speech, touch, manual gestures, etc. Thus, a comprehensive command or a meta-message is generated by the system and sent to multimedia output devices. A system-centered definition of multimodality is used in this paper. The multimodality conveys two striking features that are relevant to the software design of multimodal systems:

• the fusion of different types of data from different Input devices, and

• the temporal constraints imposed on information processing from/to Input/Output devices.

Since the first rudimentary but pertinent system, "Put That There" [1], which processes speech in parallel with manual pointing, different multimodal applications have been developed [2, 3, 4]. Each application is based on a dialog architecture combining modalities to match and elaborate on the relevant multimodal information. In such elaborations, projects usually begin from scratch and are generally based exclusively on the experiences of the designers. Consequently, they remain replications of previous results and limited synergy among parallel ongoing efforts. Today, there is no agreement on generic architectures that reflects a dialog implementation, independently of the application type. The main objective of this paper is to propose a generic architecture to analyze and extract the collective and recurrent properties, implicitly used in such dialogs.

This paper presents architectural paradigms for multimedia multimodal fusion. These paradigms use the agent architectural concept to achieve their functionalities and unify them

into generic structures. The modular structure of the proposed architecture allows easy monitoring of the global template.

The next Section summarizes the real time requirements of such dialog architectures for multimedia multimodal applications. Section 3 presents generic multi-agent architectures based on the previous analysis. Section 4 discusses the use of timed Colored Petri Networks (CPN) to model such architecture. Section 5 illustrates the proposed architecture with a stochastic timed CPN [5, 6] example of the classical "Copy and Paste" operations. Simulation tests are processed using Design CPN Tool Kit [7].

2 Multimodal Dialog Architectures: Overview and Requirements

With the increasing complexity of multimedia applications, a single modality becomes insufficient to allow the user to interact effectively across environments. A basic multimedia multimodal system as shown in Figure 1, offers the user the possibility to decide which modality or combination of modalities are better suited, depending on the task and environment contexts (see examples in [8, 9]).

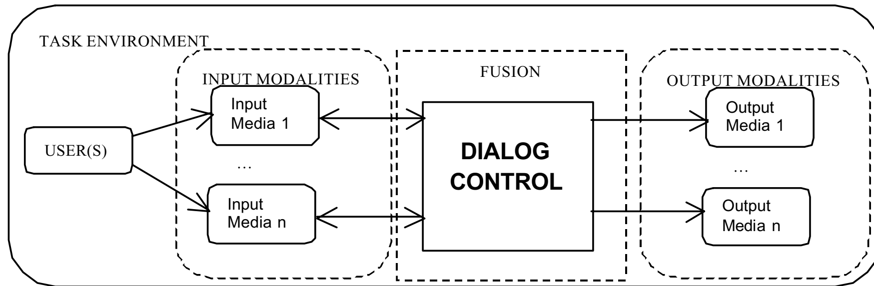


Fig. 1. *Basic multimedia multimodal model (↔: interaction, →: action).*

The environmental conditions could lead to more constrained architectures that have to be adaptable during the continuous change of either external perturbations or the user's actions.
In this context a first framework is introduced in [10] to classify interactions. It considers two dimensions ('engagement' and 'distance') and decomposes the user/system dialog into four types.

Engagement	Distance	Type of System
Conversation	Small	high-level language
Conversation	Large	low-level language
model world	Small	direct manipulation
model world	Large	low-level world

Table 1. *Interaction Systems.*

The 'engagement' characterizes the depth implication of the user in the system. The user feels that an intermediary subsystem performs the task, in 'conversation' case, and that he can act directly on the system components in 'model world' case. The `distance' represents the user cognitive effort taken.
This framework reaches the idea that two kinds of multimodal architectures are possible [11]. The first one makes fusions based on feature signal recognition. The recognition process steps of one modality guide and influence the other modalities in their own recognition steps [12, 13]. The second architecture uses individual recognition systems for each modality. Such systems are associated with an extra process that performs semantic fusion of the individual recognized

signal elements [1, 3, 14]. A third hybrid architecture is possible by mixing the two previous types: signal feature level and semantic information level.

At the core of multimodal system design, the information fusion of the input modes is the main challenge. The input modes can be equivalent, complementary, specialized or redundant as described in [15]. In this context, the multimodal system designed with one of the previous architectures (features or/and semantic levels) needs the integration of the temporal information. Figure 2 (left) shows the possible type of multimodality depending on the time proximity of the input signals. Time granularity is an important decision criterion when we generate a multimodal semantic sequence as shown in Figure 2 (right). In this example, it shows that the chosen multimodality type, for mouse clicks and speech, is the synergistic one. This is obvious in the example, because the click occurs only during the time when a sentence is said. The synergistic mouse/speech actions correspond to one statement and the tactile screen action to another one. Both statements are performed in parallel and could be independent, equivalent, complementary, specialized and/or redundant.

In other words, the temporal aspect in multimodal architecture does not only handle signals overlapping. It helps to decide whether two signal parts should belong to a multimodal fusion set or whether they should be considered as separate modal actions. Therefore, multimodal architectures are better able to avoid and recover errors that mono-modal recognition systems can't recover [14, 11]. This property results in a more robust, natural, human-machine language. Another property is that, the more timed combinations of signal information or semantic multi-signal grow, the more equivalent formulations of the same command are possible. For example, ["Copy that there"], ["copy" (click) "there"] and ["copy that" (click)] are various ways to represent three statements of a same command (copying a object in place), if speech and mouse clicking are used. This redundancy also increases the robustness in terms of error interpretation.

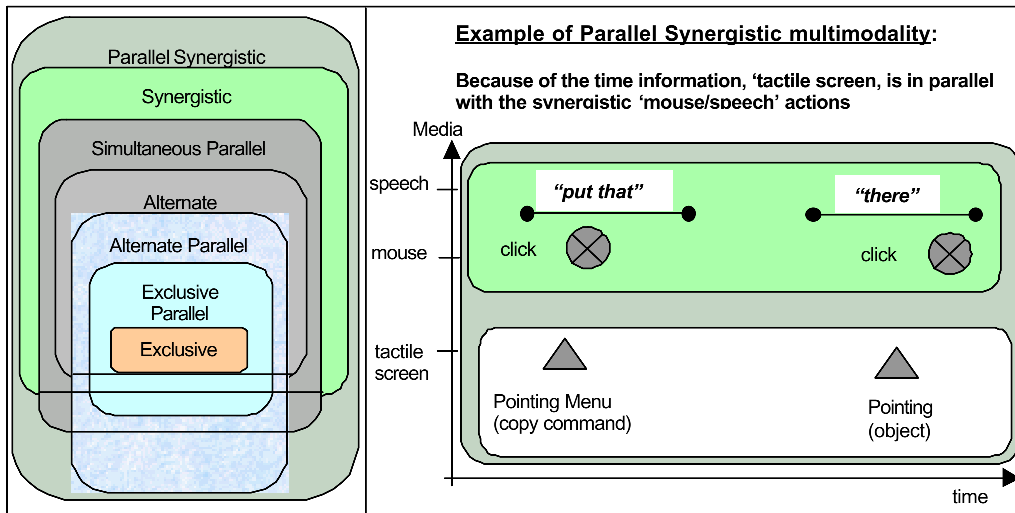


Fig. 2. *Inclusion relations between types of multimodality (left), and a three timed media example (right) [25].*

Figure 3 summarizes the main requirements and characteristics needed in multimodal dialog architectures. As shown in this figure, five characteristics can be used in the two different levels of the fusion operations: the 'early fusion' at the feature fragments level and the 'late fusion' at the semantic one [11].

The Asynchronous property gives the architecture the flexibility to handle multi external events while parallel fusions are still processing. The specialized fusion operation deals with an attribution of a same modality to a same statement type. (For example, in drawing applications, speech is specialized for color statements and pointing for basic shape statements.)

The granularity of the semantic and statistic knowledge depends on the media nature of each input modality. This knowledge leads to important functionalities. It lets the system accept or refuse the multi input information for several possible fusions (selection process); and it helps the architecture choose, between several fusions, the most suitable command to execute or message to send to an output media (decision process).
The property of parallelism is, obviously, inherent to such applications involving multi inputs.

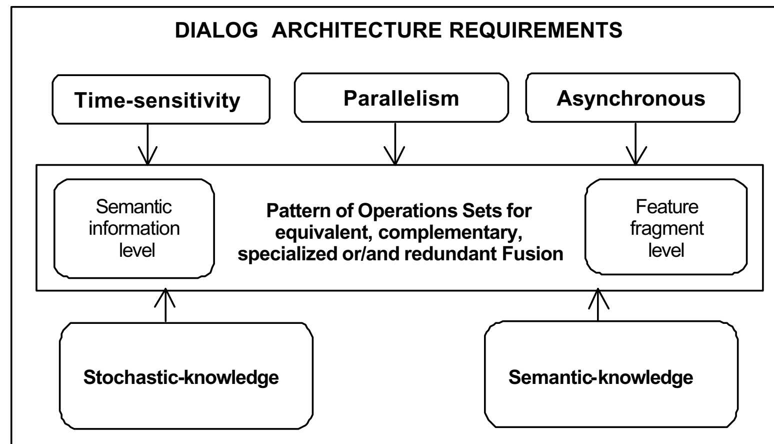


Fig. 3. *The main requirements for multimodal dialog architecture (→: used by).*

The whole requirements suggest strongly intelligent multi-agent architectures, which are the purpose of the next section.

3 Generic Multi-Agent Architectures

The Agents are entities that can interact and collaborate with dynamic and synergy for modality combination issues. The interactions should occur between agents and agents should also get information from users. An intelligent agent has three properties. It reacts in its environment at certain times (reactivity), takes initiatives (pro-activity) and interacts with other intelligent agents or users (sociability) to reach goals [16, 17, 18]. Therefore each agent could have several input ports to receive messages and/or several output ports to send ones.
The level of intelligence of each agent varies according to two major options coexisting today in the field of Distributed Artificial Intelligence [19, 20, 21]. The first one, corresponding to the cognitive school, attributes the level to the cooperation of very complex agents. This approach deals with agents with strong granularity assimilated to expert systems.
In the second school the agents are simpler and less intelligent but more active. This reactive school presupposes that it is not necessary to each agent to be individually intelligent to reach an intelligent total behavior [22]. This approach deals with a cooperative team of working agents with low granularity, which can be matched to finite automate.
Both approaches can be matched to the late and early fusions of multimedia multimodal architectures.
Obviously, there are all the possible intermediaries between these options of multi-agent systems. One can easily imagine systems based on a modular approach, putting sub-modules in competition, each sub-module being itself a universe of overlapping components. This word is usually employed for ‘sub-agents’.
Identifying the generic parts of multimodal multimedia applications and binding them into an intelligent agent architecture require the determination of common and recurrent communication protocols and their hierarchical and modular properties in such applications.

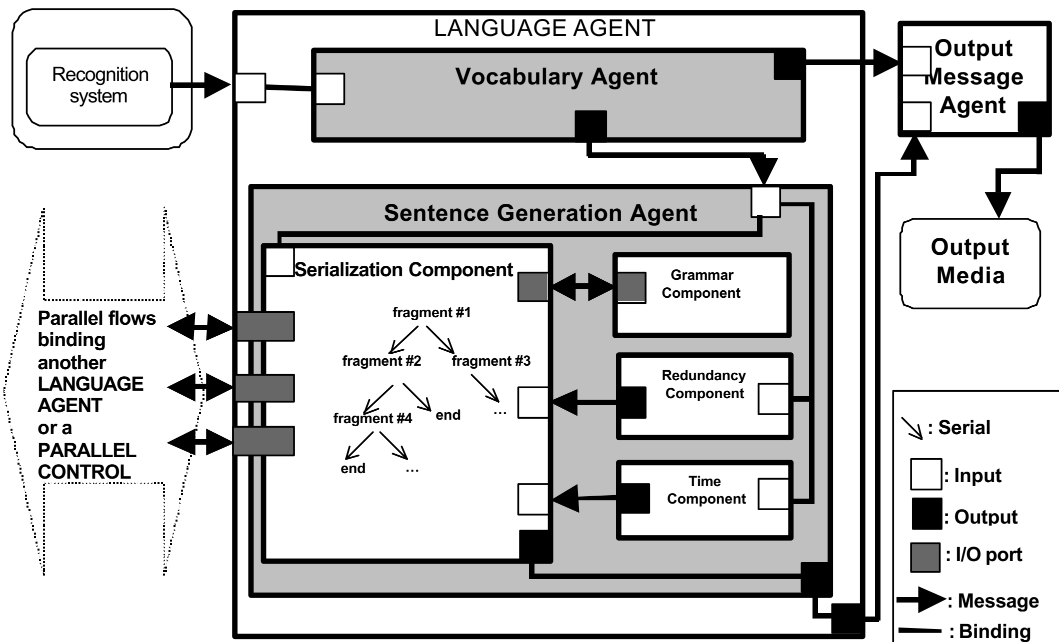


Fig. 4. *Generic Language Agent corresponding to an input modality.*

In most multimodal applications, the speech, as input modality, offers speed, a large information spectrum and relative facility of use. It lets both the user's hands and eyes free to work in other necessary tasks present, for example, in driving or moving cases. More over, speech involves a generic communication language pattern between the user and the system.

This pattern is described by a grammar with production rules, able to serialize possible sequences of the vocabulary symbols produced by users. The vocabulary could be word set, phoneme set or another signal fragment set depending on the feature level of the recognition system. The goal of the recognition system is to identify signal fragments. Then, an agent organizes the fragments in a serial sequence according to his grammar knowledge and asks others agents for possible fusion at each step of the serial regrouping. The whole interaction can be synthesized in a first generic agent architecture, as shown in Figure 4, called Language Agent (LA).

Each input modality must be associated with an LA. For basic modalities like manual pointing or mouse clicking, the complexity of the LA is strongly reduced. The 'Vocabulary Agent' that checks whether or not the fragment is known, is, obviously, no longer necessary. The 'Sentence Generation Agent' is also reduced into a simple event thread whereon another external control agent could possibly make parallel fusions. In such a case, the external agent could handle 'Redundancy' and 'Time' information, with to corresponding components. These two components are agents that, respectively, check redundancies and time neighborhood of the fragments during their sequential regrouping (Figure 4). The 'Serialization Component' processes this regrouping. Thus, depending on the input modality type, the LA could be assimilated to an expert system or to a simple thread component.

Two or more LAs can communicate directly for early parallel fusions or, through another central Agent, for late ones (Figure 5). This central agent is called Parallel Control Agent.

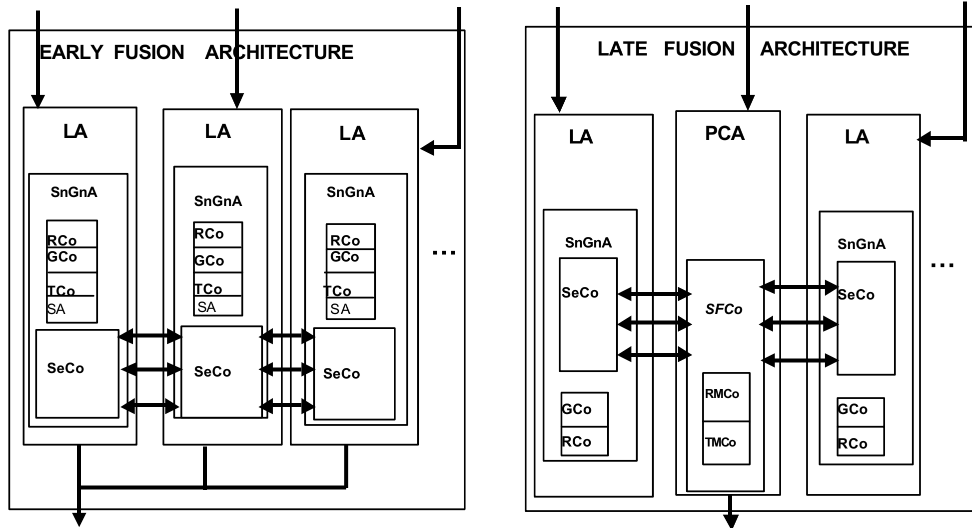


Fig. 5. *Principles of early and late fusions architectures (L: language, P: parallel, C: control, A: agent, G: grammar, S: semantic, Sn: sentence, Gn: generation, F: fusion, Se: serialization, Co: component, T: time, R: redundancy and M: management). More connections (arrows) could be added or removed by the agents to gather fusion information.*

In the first case, the 'Grammar Component' of one of the LAs must carry an extra semantic knowledge for the parallel fusion purpose. This knowledge could also be distributed between the LA's 'Grammar Components', as shown in Figure 5 (left). Several Serializing Components share their common information until one of them gives the sequential parallel fusion output. In

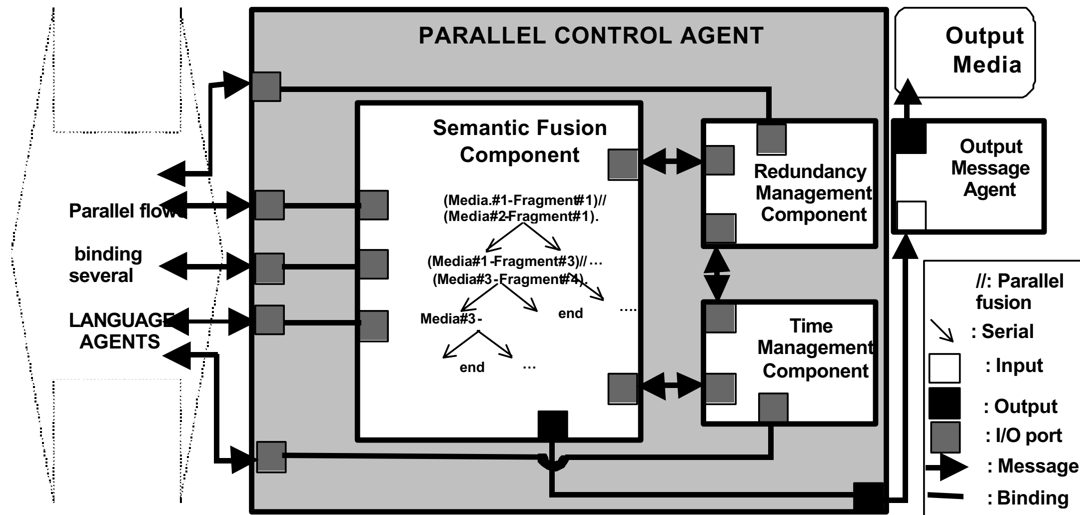


Fig. 6. *Generic Parallel Control Agent for central parallel multimodal fusions.*

the other case (Figure 5 right), a 'Parallel Control Agent' (PCA) handles and centralizes the parallel fusions of different LA information. For this purpose, the PCA has two intelligent components for, respectively, Redundancy and Time managements (Figure 6). These agents exchange information with other components to elaborate the decision. Then, generated authorizations are sent to the Semantic Fusion Component (SFCo). Based on these agreements,

the SFCo carries the steps of the semantic fusion process. As shown in Figure 6, Redundancy and Time Management Components receive the redundancy and time information via the Semantic Fusion Component or directly from the LA, depending on the complexity of the architecture and the designer choices.
The paradigms proposed in this section constitute an important step in the software development of multimodal user interface (Figure 7). Another important phase of the software development, for such

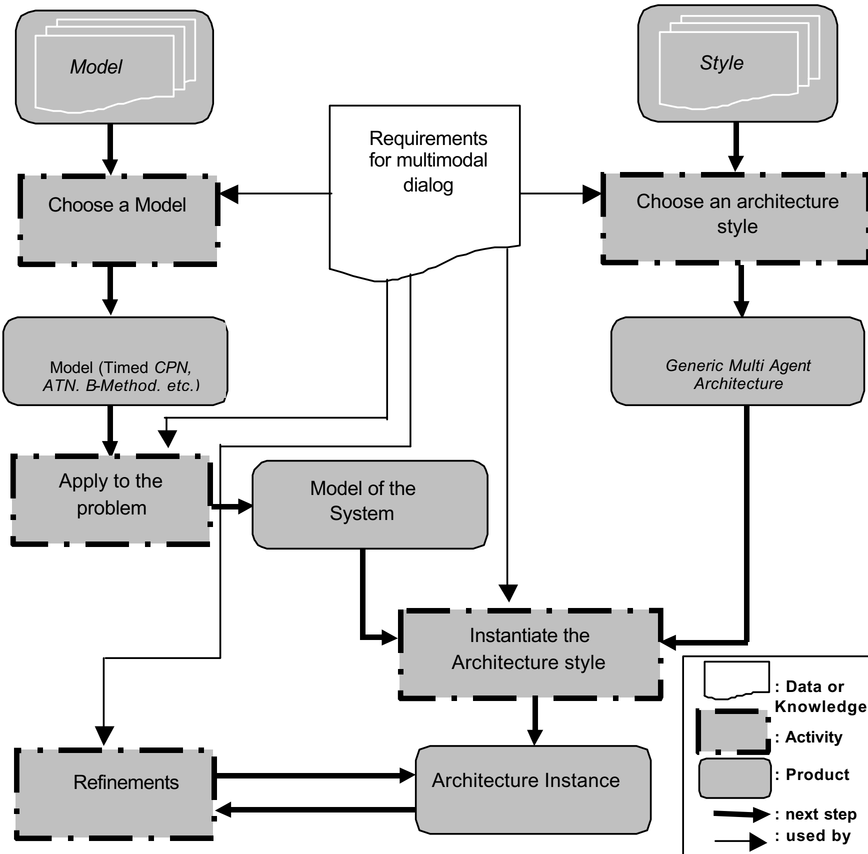


Fig. 7. *The software development phases of multimedia multimodal dialog architecture. (ATN: Augmented Transition Nets)*

applications, concerns the modeling aspect. Different methods like B_method [24], Augmented Transition Networks [25], or Timed CPN [6, 7], can be used to model the multi-agent dialog architectures. The next Section discusses the choice of Colored Petri Networks to model these architectures.

4 Petri Nets Modeling

Recently small augmented finite-state machines like augmented transitions networks have been used in the multimodal presentations system [26]. These networks easily conceptualize the communication syntax between input and/or output media streams. However, they have limitations when important constraints such as temporal information and stochastic

behaviors need to be modeled in protocols of fusion. Timed Stochastic Colored Petri Networks offer a more suitable pattern [5, 6, 7] to design such constraints in multimodal dialog.

For modeling purpose, each input modality is assimilated to a thread where signal fragments flow. Multimodal inputs are parallel threads corresponding to a changing environment that describes different internal states of the system. Multi-agent systems are multi-threaded: each agent has a control on one or several threads. Intelligent agents observe the states of one or several threads for which it is designed. Then, the agents execute actions that modify the environment. In a more formal way [17],

if $$\mathbf{A = \{a_1; a_2; \ldots\}} \qquad (1),$$

and $$\mathbf{O = \{ o_1; o_2; \ldots \}} \qquad (2),$$

are the sets of actions and observations of an agent, respectively

and if, $$\mathbf{S = \{ s_1; s_2; \ldots \}} \qquad (3),$$

is the set of states with which the environment is described (including intermediary states), then the Petri network models two kind of activities described by the functions

$$\mathbf{Observation_function : S \quad O} \qquad (4),$$

$$\mathbf{Environment_function : SxA \quad 2^S} \qquad (5).$$

The first function describes what an agent observes, in a certain state $\mathbf{s_i}$. The second one describes how the environment develop the state $\mathbf{s_i}$ when an action $\mathbf{a_i}$ is executed.
The Petri network models also the actions of the agents described by the function

$$\mathbf{Action_function : O \quad A} \qquad (6).$$

The characteristic behavior of an agent action in an environment is the set 'History':

$$\mathbf{History} = \mathbf{\{ h_1, h_2, \ldots h_i, \ldots \}} \qquad (7)$$

of all sequences of the observations defined by

$$\mathbf{h_i: (s_0)} \text{—}a_0\text{—} \mathbf{(s_1)} \text{—}a_1\text{—} \ldots \mathbf{(s_i)} \text{—}a_i\text{—} \ldots \qquad (8)$$

with $$\mathbf{a_i = Action_function\ (<s_0, \ldots, s_i>)}, \quad \forall\ i \qquad (9)$$

and $$\mathbf{s_i = Environment_function\ (s_{i-1}, a_{i-1})}, \forall\ i, i \neq 0 \qquad (10)$$

($\mathbf{s_0}$ is the initial state of the system).

To summarize the precedent transaction, the Petri network has to model the functions (4), (5), (6) and also the input media threads with the design CPN toolkit [7]. In the following, it is assumed that this toolkit and its semantics are known. The Petri network is a diagram flow of interconnected places (or locations represented by ellipses) and transitions (represented by boxes). Labeled arcs connect places to transitions. The CPN is managed by a set of rules. The rules determine when an activity can occur and specify how its occurrence changes the state of the places by changing their colored marks. The set of colored marks in all places before an occurrence of the CPN is equivalent to an observation sequence of a multi-agent system. Each

mark is a symbol that could represent signal fragments (pronounced words, mouse clicks, etc.), serialized or associated fragments (comprehensive sentences or commands) or simply a variable. In CPN each mark can be of all the data types generally available in a computer language: integer, real, string, Boolean, list, record and so on. These types are called colorsets. Thus, a CPN is a graphical structure with associated computer language statements. A transition represented by a box can model an agent. The observation function of an agent is simply modeled by input arc inscriptions and also by the conditions in each transition guard (symbolized by **[conditions]** under a box in the example represented figures 12). Input arc inscriptions specify data that must exist for an activity to occur. When a transition is fired (an activity occurs), a mark is removed from an input place and the transition activity modifies the data associated to the mark and thereby changes the state of the system (by adding a mark in an output place). If there are colorset modifications to perform, they are executed by a program associated to the transition (The program is written in a dashed line box close to the concerned transition as shown in figures 12 (a) and (b) and the symbol **C** specifies that a code is attached to the transition). Also, output arc inscriptions specify data that will be produced if an activity occurs. Thus, CPN provide an extremely effective dynamic modeling paradigm. In summary, the set of colored marks in all places before an occurrence of the net is equivalent to an observation sequence of a multi-agent system. A transition represented by a box can model an agent as shown in the examples of Figure 8.

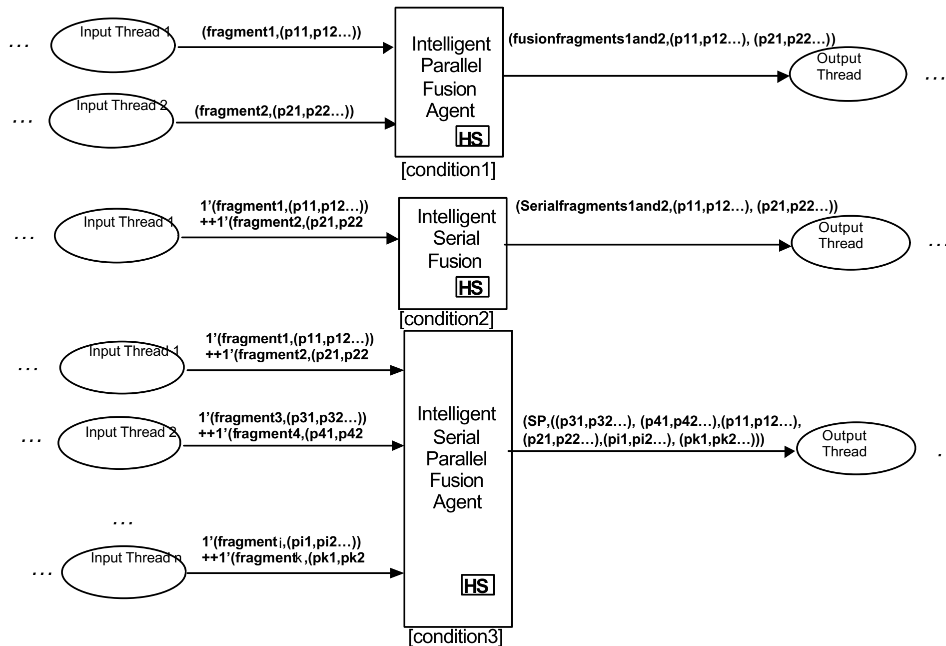


Fig. 8. *Principle of parallel, serial and serial→parallel fusions modeled by Petri Nets.*

In Figure 8, the variables, like 'p11', 'p12', etc (beginning with the character 'p'), are used to represent the time, grammatical and semantic informations of the signal fragments. The next section shows the use of variables in a practical example. The observation function of an agent is simply modeled by the conditions in each transition guard. The transition activity modifies data and thereby changes the states of the system.

5 Simulation example

In this section a typical example of a distributed architecture is presented. The 'Copy and Paste' architecture chosen involves a high level LA, for speech modality, linked, by a PCA, to a rudimentary mouse clicking LA. Tables 2 and 3 give the vocabulary, used by the speech LA, and the basic corresponding grammar. Each word has a label used in the network design.

Word	Word label
open	1
close	2
delete	3
copy	4
paste	5
cancel	6
that	7

Table 2. *Vocabulary*

Symbolic regular expressions are used to represent semantic elements. These expressions use the arrow operator for sequential concatenation in the time domain. For example, in the semantic expression:

$$(\text{word } 1 \rightarrow \text{word } 2)$$

word 1 is simply followed by (or contiguous to) word 2. In the following table, the codes (last column) are simply obtained by summing the word labels of each semantic code. The obtained codes give information used by the speech LA for serial constructions of sentences.

Set of Sentences	Command meaning	Set of corresponding semantic codes	Set of corresponding codes
{ (open→that); (open) }	Open object	{ (1→7); (1) }	{ (8); (1) }
{ (close→that); (close) }	Close object	{ (2→7); (2) }	{ (9); (2) }
{ (delete→that); (delete).}	Delete object	{ (3→7); (3) }	{ (10); (3) }
{ (paste) }	Past last copied object	{ (5) }	{ (5) }
{ (copy→that); (copy) }	Copy object	{ (4→7); (4) }	{ (11); (4) }
{ (cancel)}	Cancel last command	{ (6) }	{ (6) }

Table 3. *Grammar of authorized sentences.*

The word 'cancel' is a command that automatically cancels the last action among the authorized sentences. Therefore, if the user says one of the words labeled in the set {1, 2, 3, 4, 5} just after "cancel", the time proximity between the two words is the decision criterion for suppressing the second word or taking it as a next command. For the proposed architecture both scenarios are processed.Non-authorized sentences used in this architecture are given in Table 4.

Non authorized Sentences	Corresponding semantic codes	Corresponding codes
(that), (that→…)	(7), (7→…)	(7), (8)…(13)
(paste→that)	(5→7)	(12)

Table 4. *Grammar of non-authorized sentences and their codes.*

The multimodal dialog gives for each sentence a set of possible redundant fusions. The symbol // models these concurrent associations in regular expressions. For example, depending upon

temporal information, the first command given in Table 3 is an element of the following semantic fusion set:

{(click→open→that); (open→click); (click→open); (click // open); ((click // open)→that); (click // (open→that))}.

This semantic set includes the grammatical sentences corresponding to the command 'Open object'. Words temporally isolated and labeled in the set **{1, 2, 3, 4, 7}** are not considered by the PCA. The remaining fusion entities like **((close→open) // click)**, **(click // (delete→open))**, etc. or isolated clicks are also ignored by the system. The whole sets constitute the semantic knowledge. The main focus in this paper is how to use a timed semantic knowledge to achieve a multimodal fusion.The global declaration page, used in the timed CPN example, is shown in Figure 9 below.

```
                                   (*GLOBAL DECLARATION PAGE*)
                                  (*==========================*)
 (*Proximity time between two events*)
(*==========================*)
   val ProxyTime = 100;
(*Average Inter_arrival*)
(*================*)
   val ClickArrival = ref 1.0;
   val WordArrival = ref 10.0;
 (* Color sets *)
(*========*)
  color Int =int;
  color Attribute = product Int * Int * Int;
          (* Color sets for mouse event *)
          (************************)
          color MouseClick = with ClickEvent;
          color ClickxAttribute = product MouseClick * Attribute timed;
          color ME = with me timed;
           (* Color sets for speech*)
          (*******************)
           color WordSaid = with Word;
          color WordxAttribute = product WordSaid * Attribute * Int timed;
          color WE = with we timed;
           (* Color sets for fusion event*)
          (************************)
          color FusedEvents = with Fused;
          color FAttrib =product Int * Int;
          color FusedxAttribute = product FusedEvents * FAttrib * FAttrib * FAttrib timed ;
(*Variables*)
(*=======*)
var Word1, Word2,Word3 :WordSaid;
var n, Fn, Fm, Fm1, Fm2, Fm3, m, m1, m2, m3, p, Fp : Int;
          (* Variables for Time *)
          (******************)
          var NextClick, NextWord, ArrivalTimeC, ArrivalTimeW1, ArrivalTimeW2,
          ArrivalTimeW3, ArrivalTimeWp, ArrivalTimeW: Int ;
          (* Variables for word labels *)
          (**********************)
          var wt, wtype, wtype1, wtype2, wtype3 : Int;
(* Functions *)
 (*========*)
fun intTime()=IntInf.toInt(time());
fun round x =floor(x+0.5);
fun ExpLaw x= round(exponential(x));
```

Fig. 9. *Global declaration page.*

Also, a place filled with a pattern symbolizes a thread with input and output ports, as shown in Figure 10.

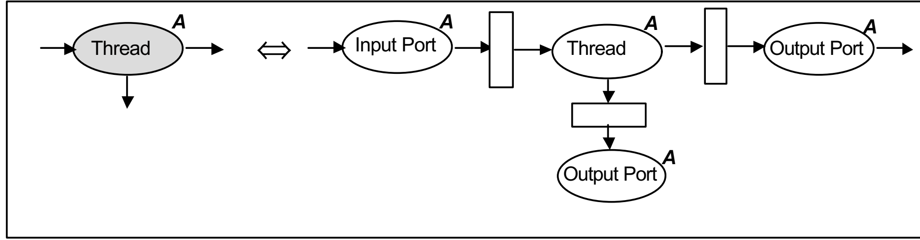


Fig. 10. *Symbolic representation used in figures 12 (a) and (b).*

Input (P In) and output (P Out) ports of the Petri network correspond as well to the architecture ones (see Figures 4 and 6). The associated network, as shown in Figures 12 (a) and (b), uses two random generators to design the arrival time of the input media events. The generators are drawn in the top of Figure 12 (a) with dashed non-bold lines and both are modeled with the transitions named '**Click**' and '**WordSaid**'. The inter-arrival time between two pronounced words as well as the time between two consecutive 'clicks', are exponentially distributed. Events (like words and clicks) are generated or arrived in two different threads (the places '**ThreadofClick**' and '**ThreadofWords**'). The time between two click (respectively word) arrivals has a mean = **ClickArrival** (respectively = **WordArrival**). The inter-arrival time between 2 click (respectively word) events has an exponential distribution with parameter r =1/**ClickArrival** (respectively 1/**WordArrival**). (Mean: 1/r and Variance: $1/(r^2)$). The density function of the inter-arrival time between 2 events is $f(x) = r * \exp(- r * x)$, if x is greater than 0 and $f(x) = 0$ elsewhere. The inter-arrival time follows an exponential law, for the words and also for the clicks. If the proximity time between a word event and a click event is below **ProxyTime** and if these two events verify the grammatical and semantic conditions then these two events are fused into one command. The transitions drawn with bold dashed lines model the PCA components distributed over the network. Transitions, with bold lines, model the speech LA components in Figures 12 (a) and (b). The mouse click LA is reduced to a simple thread: **'ThreadofClicks'**. The figures 11 (a), (b) and (c) show the simulation results for **WorArrival**=**ClickArrival**=5000ms and **ProxyTime**=10000ms. Figure 11 (c) presents the number of achieved fusions in the time (or the number of marks in the place '**FusionedMedia**' of the CPN). In the same way, a command can be cancelled if the user says the word 'cancel' just after an achieved command (the proximity time between the two events: the command and the word 'cancel' is chosen below (**ProxyTime**/25)). Figure 11 (b) shows the accumulation of words in the corresponding thread (or the number of marks in the place '**ThreadofWords**'). Figure 11 (a) shows the resulting cancelled commands in the time (or the number of marks arrived in the place '**CanceledCommand**'). The transition '**RecognitionSystem**' (Figure 12 (a)) assigns a random label '**wtype**' to each word present in the place '**WaitRecognition**'. This random assignation does not model a real flowing speech because automatic modeling of user speech is outside the scope of this paper.

The symbol HS in LA transition means that such transition is a Hierarchical Substitution ones. The network in Figure 12 (b) describes interactions at a sublevel of the network in Figure 12 (a).

The symbol FG in identical places indicates that the places are 'global fusion places' [7]. These identical places are simply a unique resource shared over the net by a simple graphical artifact: the representation of the place and its elements is replicated with the symbol FG.

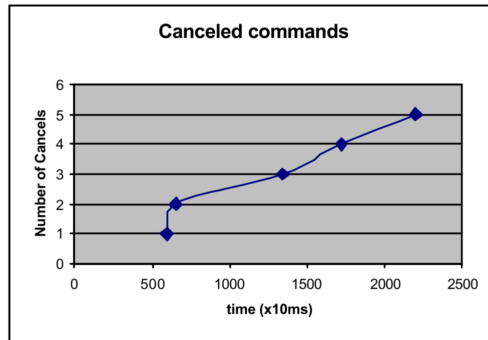


***Fig. 11. (a)** Simulation Results: Canceled commands*

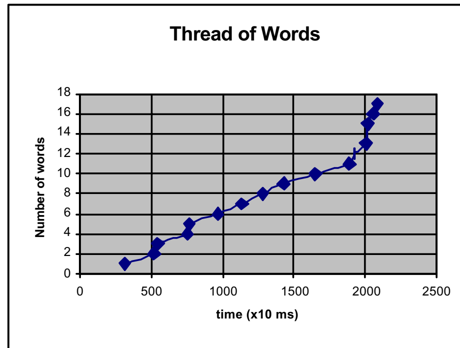


***Fig. 11. (b).** Simulation Results: Thread of words.*

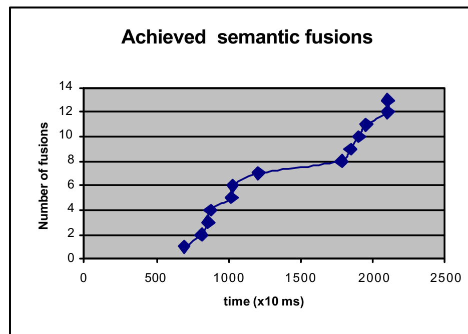


***Fig. 11. (c)** Simulation Results: Achieved fusions.*

Modeling with timed colored Petri nets leads multiple advantages. First of all, These nets can validate a model of timed fusion engine (as shown by the results in figures 11 (a), (b) and (c)). The arrival time between two consecutive events and the processing time by an agent are adjustable. These settings can follow laws of probability (exponential, Erlang etc.). Secondly, the nets are suitable to a distributed modeling where each transition assumes the function of a specialized agent. The function of each agent is easily modifiable (by changing the guard

conditions or the code associated to the transition). Besides, modeling a 'fission' (opposite process of a fusion) is simple to implement because the colorsets of fused marks can keep all the information for that purpose. For example, on Figures 12 (a) and 12 (b), the input arc inscription (to the '**FusionedMed** place) has event numbers **m** and **n** corresponding to pronounced word and click events respectively. Finally processing concurrent independent tasks is easy.

6 Conclusion

In this paper we have proposed a new agent based architectural paradigms for multimedia multimodal fusion purpose. These paradigms lead to new generic structures that unify the several applications in multimedia multimodal dialog. They also offer to developers a framework specifying different functionalities used in multimodal software implementation. In a first phase, we have gathered the main common requirements and constraints that multimodal dialogs need. We have then identified two interaction types related to the early and late fusions. After identifying the generic recurrent characteristics shared by all modalities, each input media is associated to a specific Language Agent (LA). The LAs are interconnected directly or through a Parallel Control Agent (PCA) to perform the dialog. The architecture of the PCA is decomposed into intelligent components, which provides a modular structure. These components manage temporal, redundant and grammatical conflicts. The proposed architectures are modeled with timed Colored Petri Networks and support both parallel and serial fusions. A typical simulation example, with random input flows, has illustrated our approach. The simulation allowed us to verify that the proposed architectures performed correctly the expected fusions.

Acknowledgments : We wish to acknowledge the financial support of the Natural Sciences and Engineering Research Council (NSERC) of Canada

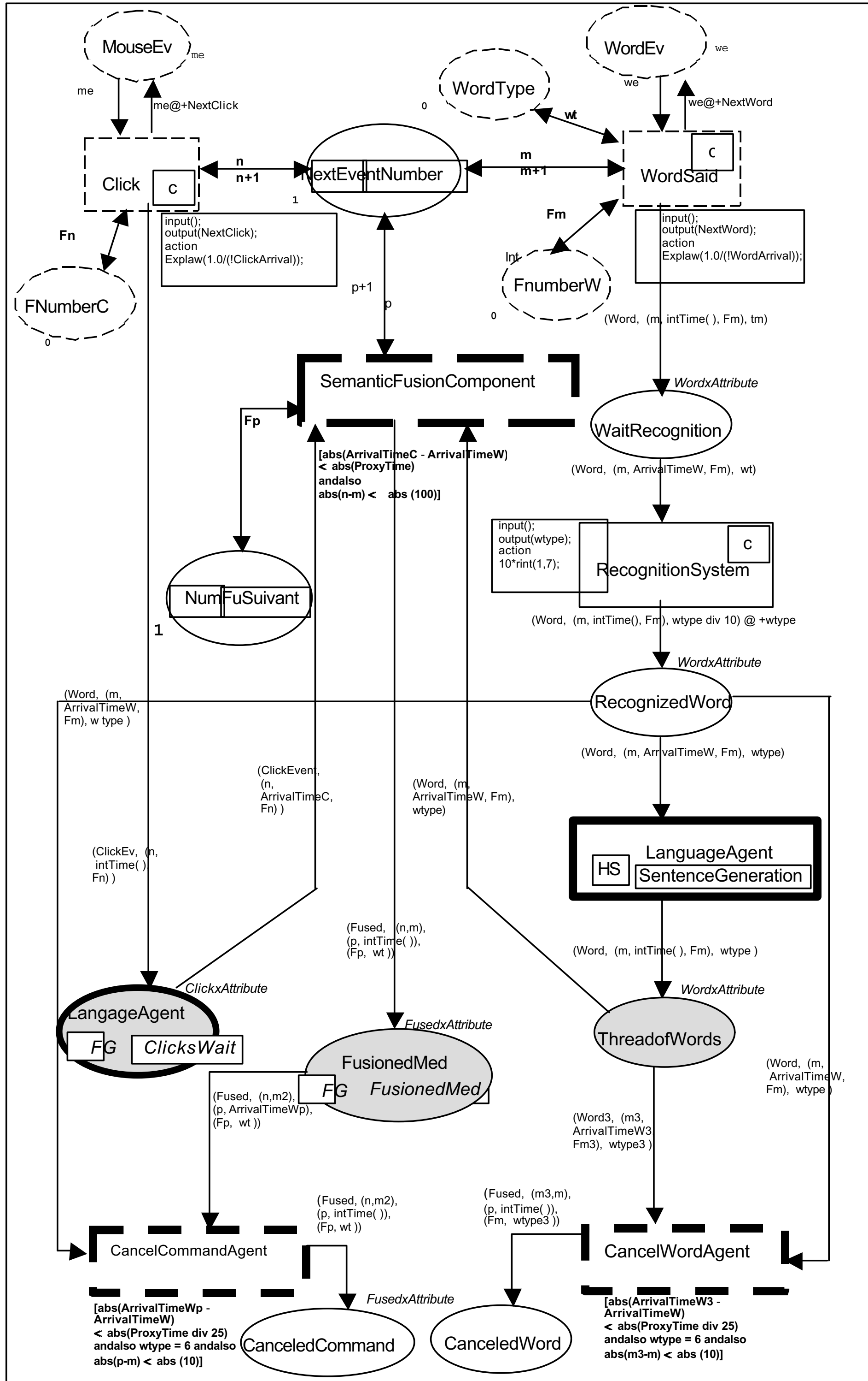


Fig. 12. *(a). Bimodal fusion dialog.*

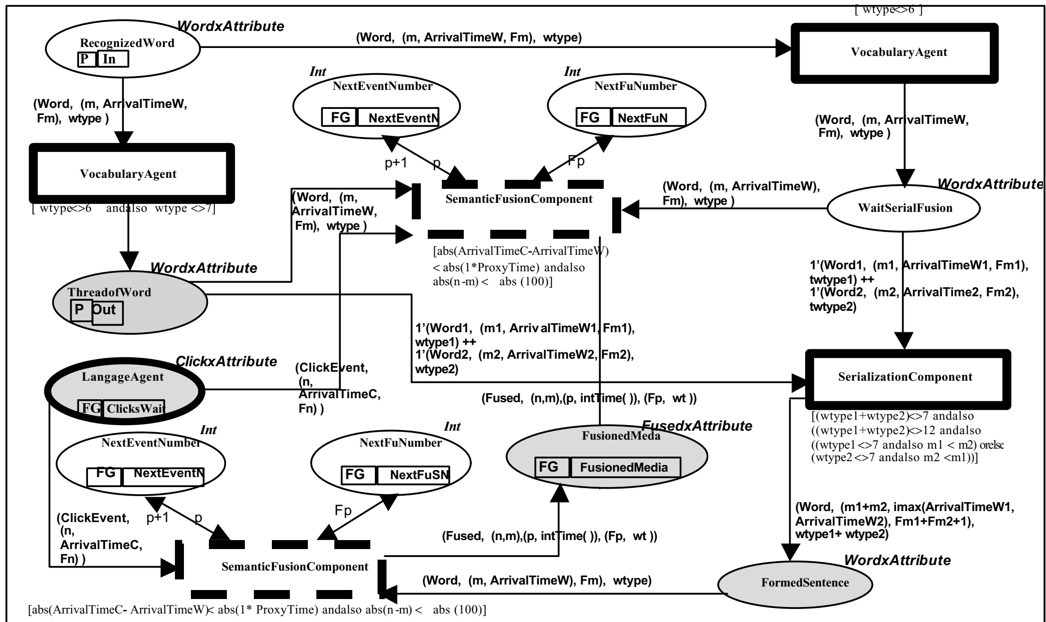


Fig.12 (b). Bimodal fusion dialog (Sub level).

REFERENCES

1. Bolt, R.A., *Put that there: Voice and gesture at the graphics interface,* ACM Computer Graphics 14,3, 262-270, 1980.
2. Crowley, J.L. and Bérard, F. *Multimodal tracking of faces for video communications,* In Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR'97), San Juan, IEEE Press, NY, June, 1997.
3. Bellik Y., Burger D., *Multimodal Interfaces: New Solutions to the Problem of the Computer Accessibility for the Blind.* Proc. CHI'94, Boston, 24-28 April 1994.
4. McGee, D.R., Cohen, P.R., and Wu, L., *Something from nothing: Augmenting a paper-based work practice with multimodal interaction*, in the Proceedings of the Conference on Designing Augmented Reality Environments, ACM Press, Helsingor, Denmark, 71-80, April 12-14, 2000.
5. Jensen, K., *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing, ISBN: 3-540-60943-1, 1997.
6. Jensen, K., *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing, ISBN: 3-540-58276-2, 1997.
7. Jensen, K., Christensen, S., Huber, P and Holla, M., *Design/CPN Reference Manual,* Department of Computer Science, University of Aarhus, Denmark, http://www.daimi.au.dk/designCPN/, 1995.
8. Oviatt, S.L., *Multimodal Signal Processing in Naturalistic Noisy Environments,* In B. Yuan, T. Huang and X. Tang Eds., Proceedings of the International Conference on Spoken Language Processing (ICSLP'2000), Vol. 2, pp. 696-699, Beijing, China: Chinese Friendship Publishers, 2000.
9. Oviatt, S.L., *Multimodal System Processing in Mobile environments,* Proceedings of the Thirteenth Annual ACM Symposium on User Interface Software Technology UIST'2000, pp. 21-30, New York: ACM Press, 2000.
10. Hutchins, E. L., Holland, J. D. and Norman, D. A., *Direct manipulation interfaces,* In Norman, D. A. and Draper, S. W. Eds., User centred system design: new perspectives on human computer design, Hillsdale, NJ, Lawrence Erlbaum, 1986.
11. Oviatt, S.L., Cohen, P.R., Wu, L., Vergo, J., Duncan, L., Suhm, B., Bers, J., Holzman, T., Winograd, T., Landay, J., Larson, J. and Ferro, D., *Designing the user interface for multimodal speech and gesture applications: State-of-the-art systems and research directions*, Human Computer Interaction, vol. 15, no. 4, pp. 263-322, 2000.
12. Bregler, C., Manke, S., Hild, H., and Waibel, A. *Improving connected letter recognition by lip reading,* Proceedings of the International Conference on Acoustics, Speech and Signal Processing, 1, pp. 557-560. IEEE Press, 1993.
13. *Projet AMIBE*, Rapport d'activité, GDR |n° 9, GDR-PRC Communication Homme-Machine, CNRS, MESR, 1994, pp. 59-70 (Technical Report in french), 1994.
14. Oviatt, S. L., *Mutual disambiguation of recognition errors in a multimodal architecture*, Proceedings of Conference on Human Factors in Computing Systems: CHI '99, New York, N.Y., ACM Press, 576-583, 1999.
15. Coutaz, J., Nigay, L., *Les propriétés CARE dans les interfaces multimodales,* IHM'94. Sixièmes journées sur l'ingénierie des Interfaces Homme-Machine, Lilles, 8-9 Déc, (French paper), 1994.
16. Jennings, N. R. and Wooldridge, M. J., "*Applications of Intelligent Agents*" in "Agent Technologies: Foundations, Applications and Markets", Eds. N. R. Jennings and M. Wooldridge, 3-28, 1998.
17. Weiss, G., *Multiagent Systems*, MIT-Press Ed., 1999.

18. Bird, S.D., *Toward taxonomy of multi-agents systems,* International Journal of Man-Machine Studies, 39, 689-704. 1993.
19. Bond, A.H. and Gasser, L., *Readings in Distributed Artificial Intelligence,* San Mateo, Calif.: Morgan Kaufmann, 1988.
20. Ishida, T., *Real-Time Search for Learning Autonomous Agents*, Kluwer Academic Publishers, 1997.
21. Muller, H. J., *Negotiation principles*, In G. M. P. O'Hare and N. R. Jennings, eds, Foundations of Distributed Artificial Intelligence, pp. 211-229, Wiley, 1996.
22. Cohen, P. R., Levesque, H. R., and Smith, I., *On team formation,* Hintikka, J. and Tuomela, R. (Eds.) Contemporary Action Theory. Synthesis, 1997.
23. Tambe M., Johnson W. L., Jones E. M., Koss F., Laid J. E., Rosenblum P. S., and Schwamb K., *Intelligent agents for interactive simulation environments*, Al Magazine, 16(1), pp. 15-39, 1995.
24. Abrial J.-R, *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
25. Bellik Y., : Thèse de Doctorat de l'Université de Paris XI, spécialité informatique.*« Interfaces multimodales : concepts modèles architectures »* soutenue le 30Mai 1995 par Yacine Bellik PHD Thesis of univerity au ParisXI (France)
26. Chen, S.-C. and Kashyap, R. L., *Temporal and spatial semantic for multimedia presentations,* International Symposium on Multimedia Information Processing, pp. 441-446, Dec.11-13, 1997.

Towards an object based multi-formalism multi-solution modeling approach

G. Franceschinis[(1)], M. Gribaudo[(2)], M. Iacono[(3)], N. Mazzocca[(3)] and V. Vittorini[(4)] *

(1) Univ. del Piemonte Orientale, Alessandria, Italy, giuliana@mfn.unipmn.it

(2) Univ. di Torino, Torino, Italy, marcog@di.unito.it

(3) Seconda Univ. di Napoli, Aversa, Italy, {n.mazzocca,mauro.iacono}@unina2.it

(4) Univ. di Napoli "Federico II", Napoli, Italy, vittorin@unina.it

Abstract

Analysis and simulation of complex systems is an hard task that requires the use of proper modeling formalisms and tools. In many cases, no single analysis and modeling method can successfully cope with the growing complexity of a real system. A multi-formalism multi-solution approach is very appealing, since it allows to cope with the complexity of the problem by using different formalisms to model and analyze different subsystems and also to define reusable building blocks. Nevertheless problems arise at many levels. The major concerns are: the interoperability of different formalisms and analysis/simulation tools, the definition and the implementation of mechanisms to guarantee the flexibility and the scalability of the modeling frameworks and the development of proper strategies for the analysis of multi-formalism multi-solution models.

This paper describes a multi-formalism multi-solution approach to the construction of models based on the integration of different graph-based formalisms. The proposed approach is based on an object oriented construction method and it is supported by the DrawNET++ framework through a proper interface to the external solvers (analysis/simulation engines) realized by means of the XML and XSL technologies.

A simple dependability example is used throughout the paper to describe the modeling process and the possibility of analyzing a system by integrating two different formalisms: Fault Trees (FT) and Generalized Stochastic Petri Nets (GSPN).

Keywords: Object Oriented composition of models, multi-formalism modeling, Petri Nets, Fault Trees.

1 Introduction

Analysis and simulation of complex systems is an hard task that requires the use of proper modeling formalisms and tools. The complexity of modeling and analysis of real systems can be mastered through a "divide and conquer" approach: indeed modular approaches to models construction allow to cope with the complexity of models by defining libraries of reusable building blocks (sub-models) and encouraging a modeling discipline. Moreover, modular approaches enable the construction of multi-formalism models and stimulate the development of multi-solution techniques that take advantage from efficient formalism-specific solution methods.

*This work is partially supported by the MIUR (Project "ISIDE").

Some formalisms directly support composition in their definition (as Process Algebras), others do not include any support to composition/refinement of models in their original definition, but several proposals for adding these features a-posteriori have appeared in the literature [2, 17].

In the context of multi-formalism multi-solution modeling, it is interesting to observe that despite the fact that composition techniques of distinct classes of formalisms seem different, they often have several common aspects. It is thus natural to think of a framework for the composition of multi-formalism models. Some results in this direction have been achieved within the Möbius project [9]. Nevertheless, much more work should be done in this line since problems arise at many levels. The major concerns are: composition issues when integrating sub-models, the interoperability of different formalisms and analysis/simulation tools, the definition and the implementation of proper mechanisms to guarantee the flexibility and the scalability of the modeling framework.

This paper describes a multi-formalism multi-solution approach to the construction of models based on the integration of different graph-based formalisms, using an example from the area of dependability. The proposed approach is based on (a) an Object Oriented (OO) methodology and (b) a proper interface to external solvers (analysis/simulation engines). At the state of our research, the presented methodology is partially supported by the DrawNET++ framework [14, 13].

With respect to point (a), the proposed method exploits composition, facilitating the model structuring and (sub)model reuse in a style inspired by the OO paradigm. Some OO features are present also in other existing frameworks (for example Tangram-II [6]). Our proposal goes one step further. The key concepts behind our OO construction method are model metaclasses (allowing formalisms definition and inheritance), model classes, model instances (objects), weak and strong aggregation.

With respect to point (b), new graph-based formalisms can be created and easily integrated in the DrawNET++ framework without any programming effort. In particular, the nodes of a graph may represent domain specific sub-models expressed in some underlying formalism by an expert model designer, and presented to the final user as black boxes with proper interface and connectors. In order to allow the different formalisms and solution methods to inter-operate an XML description of the sub-models is used. The back end of the framework is an interface towards the solvers that consists of (1) XSL filters to translate the XML representation of the DrawNET++'s models into the formats used by the external solvers, (2) scripts for running solvers, (3) filters to feed the results back into the XML models representation.

This approach differs substantially from that advocated in the Möbius project [9] where new formalisms, composition operators and solvers are actually implemented within a unique comprehensive tool but all formalisms are required to be described in terms of a predefined general framework [10].

The paper is organized as follows. Sec. 2 briefly places our work in the context of the fault trees analysis techniques and introduces the simple example we will use throughout the paper. In Sec. 3 the OO model construction method is described and applied to the example. Sec. 3 mainly deals with multi-formalism modeling, whereas Sec. 4 presents a new multi-solution strategy developed for the class of applications of the running example which combines Fault Trees (FT) and Generalized Stochastic Petri Nets (GSPN). Finally, Sec. 5 describes the architecture of the DrawNET++ interface to solvers needed to support the FT/GSPN integrated solution approach.

2 A primer on FTs and mixed FT analysis techniques

FTA (Fault Tree Analysis) techniques are widely used to model the failure modes of dependable systems [15, 16]. A minimal cut set (MCS) of FTA shows a minimal combination of component failures (or Basic Events, BEs) leading to system failure (the Top Event, TE), i.e. to the occurrence of an undesirable event. FTA tools allow to compute all the minimal cut sets for a given FT model as well as their probability, and the probability of the TE. Such analysis methods are efficiently applicable only under quite restrictive hypothesis. The quantitative analysis of Fault Trees (FT) models assumes independence among component failures and it is based on combinatorial solution methods. State space solution methods are necessary to include more flexibility and expressive power (e.g. dependence between basic faults, repair, complex fault tolerance strategies).

Mixed solution methods are however possible, based on the concept of "minimal" independent subtree [1, 12]; in this case a state space method can be applied only to the smallest sub-models that actually need it. The result of the subtree analysis can then be fed back into the upper FT part to perform combinatorial analysis. To perform state space based analysis, a suitable formalism must be used: we have chosen GSPN, since automatic translation from FT to GSPN formalism is possible.

Complex gates may be included in this case, as well as subsystem repair facilities (with or without constraints on the number of available repair facilities). It turns out that GSPNs can be used to express the most common repair strategies, and a repair GSPN sub-model can be easily composed (using a place superposition operator) with the GSPN automatically obtained from a FT. This leads to a multi-formalism model of a dependable system, allowing to combine FT and GSPN sub-models and to apply a multi-solution method to solve the resulting model.

In this paper we will use a FT case study to present our approach which wants to exploit the advantages of an OO modular approach to the modeling of systems. In order to represent repairs at the FT level, we compose FT models with pre-defined blocks representing GSPN sub-models of repairs (i.e. Repair Blocks, RBs).

The FT example used throughout the paper is presented in Fig. 1, where (k:n) gates are used. It is the FT model of a highly redundant multiprocessor system which consists of three subsystems SUB_i, $i \in \{1, 2, 3\}$, a shared memory M_s and two buses connecting the subsystems and the shared memory.

Each subsystem SUB_i consists of a processing unit CPU_i, a local memory M_i and two disks $\mathrm{D}_{i,j}$, $j \in \{1, 2\}$.

Redundancy is adopted at the system level (through the two bus lines) and at the local level (the two disks contains the same data). Moreover, the shared memory may be used to replace each local memory if it fails. Note that, as a matter of fact, a FT model can be an acyclic graph, since one or more BEs may be common to different subtrees (like M_s in the example).

At the system level a fault occurs if the two buses fail or two out of three subsystems simultaneously fail. At the subsystem level a fault occurs in the following cases: either the CPU fails, or both the disks fail, or both the shared and the local memory fail.

The presence of a shared memory introduces a dependence among the subsystem, since the BE representing the related fault is a leaf that is common to the subtrees representing the three subsystems in Fig. 1. A slightly different version of this example will be also used in the paper where M_s is removed and a redundant local memory is added to each subsystem. In this case the subsystem fails when both its memories fail.

Figure 1: A FT model of the multiprocessor system

In the next sections, we will extend the FT model of the multiprocessor system with repair actions in order to evaluate the system unavailability when a preventive maintenance policy is implemented, e.g. the repair of a subsystem or the repair of one part of any subsystem is activated as soon as it fails.

3 The OO model construction methodology

In this section we introduce a new approach to multi-formalism modeling of systems. In order to define scalable and flexible mechanisms to integrate different formal languages we are developing a framework based on the concept of *Metaformalism* and an object oriented methodology to create models and reusable models libraries. All the concepts introduced in the following will be illustrated in the next subsections by means of the running example.

A *Metaformalism* is a language used to describe **graph-based formalism**, i.e. formalisms whose elements are nodes and edges, such as Petri Nets, Queueing Networks, Fault Trees. In other words a *Metaformalism* is a formal language that allows to easily define any graph-based formalism within our modeling framework.

According to our methodology the development of a model is accomplished by adopting an object oriented approach. Paraphrasing Booch's definition of object oriented programming [5], we want to provide "a modeling method in which a model is organized as cooperative collection of sub-models[1](objects), each of which represents an instance of some model class, and whose classes are all members of a hierarchy of model classes united

[1]Since in our approach each model can be regarded as a part of a more complex system, in the following

via inheritance relationships".

To this purpose we define a three level object oriented system:

- Level 1: Model Metaclasses. The first level addresses the formal languages to be used in the modeling framework. The formal languages are all described through the rules expressed by the *Metaformalism*. We call Model Metaclass (Metaclass for short) any formalism description expressed through the *Metaformalism*. The FT formalism is described by the FT Metaclass, since it defines the elements that can be used to build FT models.

- Level 2: Model Classes. The second level addresses the sub-models specification. A sub-model at this level is not a concrete entity existing in time and space yet. It is just an abstraction, called Model Class, which facilitates the grouping of set of objects sharing a common structure consisting of nodes and edges and a set of parameters to be defined at object instantiation time. Such structure must be compliant to its Metaclass, i.e. a Model Class is specified by means of the elements described by its Metaclass.

- Level 3: Model Objects. The third level addresses the sub-models instantiation. At this level sub-model (Model Objects) are created and used to build a complete model. A Model Object is a tangible entity completely characterized by the actual value of all the parameters specified in the structure of its Model Class, for example the firing rates of the timed transitions or the initial marking if its reference formalism is GSPN.

At the state of our research, the inheritance relation is introduced at the level of the Model Metaclasses. It is possible to build hierarchies of Metaclasses and it is also possible to build hierarchies of elements within a Metaclass. For example it is possible to define the GSPN formalism by inheriting from the Petri Net (PN) formalism, moreover is possible to define both the AND node and the OR node of the FT formalism by inheritance from the a common ancestor, the GATE node.

Metaclasses inheritance implements the well known "is a" relationship. If the Metaclass D is derived from the Metaclass B, the structure of D shares the elements defined by the structure of B. The advantage of using Metaclass hierarchies is that through the hierarchy a better abstraction is realized that allows to identify the common properties of models.

Elements inheritance inside a Metaclass is still an "is a" relationship at a lower level. An example of use is for the definition of edges that must be able to connect different types of nodes belonging to the same "family" (e.g., in the FT formalism event nodes can be connected to GATEs, independently of the specific gate type).

To support submodel composition it may be necessary to explicitly define interface elements and interface connection edges in a proper "wrapper" formalism, in particular when submodels (explicitly) defined through a formalism F1 are included into models described in a different formalism F2, then it is mandatory to define in a third "wrapper" formalism new edges that can connect elements belonging to F1 and F2. .

A complete model can be built in two different ways:

1. A **flat model** is a model built by using the formalism elements defined by a given Metaclass, e.g. a Fault Tree model obtained by using the FT Metaclass, or a GSPN

the word 'model' is often used interchangeably with the word 'sub-model'.

model obtained by using the GSPN Metaclass. A flat model may be a complete model or it can also be stored as a Model Class to be reused.

2. A **composed model** is a model containing at least one Model Object or at least one Model Class. In the former case it is usually intended to be a final model, in the latter it is intended to be a new Model Class.

   The composition (also referred as *aggregation* in the following) is accomplished by connecting the interfaces of the objects through proper edges and operators. We call this operation a "weak aggregation".

   If the composed model is built to be a new Model Class, (some of) the interface nodes that have been used to create the composed model might have to be hidden. We call this operation "strong aggregation", since it is a form of aggregation association with strong ownership and coincident "lifetime" of the contained objects as part of the whole.

The described methodology supports the multi-formalism modeling in a very natural way. Different formalisms can coexist in the same model implicitly or explicitly. An implicit multi-formalism model is defined at the Model Metaclasses level by extending the formalism definitions with *implicit nodes.* An implicit node represents a part of the model that is not defined at the specification and design time, but it contains all the information necessary to define it. In other words, an implicit node could be replaced by a sub-model expressed through a different formalism. The implicit nodes may be added into an existing formalism by Metaclass inheritance.

An explicit multi-formalism model is defined by allowing to define composed models in which Model Objects or Model Classes are included that are defined by means of different Metaclasses.

In this paper we will use the implicit multi-formalism approach by defining a Repairable Fault Tree Metaclass (RFT), inheriting from the FT Metaclass. The implicit node defined by the RFT Metaclass is a Repair node (called Repair Block in the rest of the paper) which *implicitly* defines a GSPN model of a repair action. The proposed framework however allows also the explicit version, since repair nodes may be replaced by submodels described according to a GSPN Metaclass definition.

In the following we discuss the implementation of our method within the DrawNET++ framework [14, 13].

3.1 Metaformalism and Metaclasses

DrawNET++ supports the proposed methodology since it allows to define and manipulate graph based formalisms and provides some OO features. It relies upon the XML technology: an XML description of the formalism must be defined to customize the DrawNET++ graphical editor to design models based on that formalism and generate an XML description of the model.

Thus, it is straightforward to express the Metaformalism by a Document Type Definition (DTD) and the Metaclasses by XML specifications compliant to the Metaformalism definition.

The FT Metaclass (partially) reported in Fig. 2 describes the formalism used to model the multiprocessor system introduced in Sec. 2.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE formalism SYSTEM "formalism.dtd">
<formalism parent="" name="FT">
 <propertyType name="Title" default=""/>

 <nodeType parent="" name="Event0">
  <propertyType name="Label" default=""/>
 </nodeType>
 <nodeType parent="Event0" name="Event"/>
 <nodeType parent="Event0" name="BasicEvent">
  <propertyType name="Distribution" default="ALL EXP 1.0"/>
 </nodeType>

 <nodeType parent="" name="Gate"/>
 <nodeType parent="Gate" name="And"/>
 <nodeType parent="Gate" name="Or"/>

 <nodeType parent="" name="G2of3"/>

 <edgeType parent="" name="Arc">
  <constraint fromType="Gate" fromCardinality="1"
   toType="Event" toCardinality="1"/>
  <constraint fromType="G2of3" fromCardinality="1"
   toType="Event" toCardinality="1"/>
  <constraint fromType="Event0" fromCardinality=""
   toType="Gate" toCardinality=""/>
  <constraint fromType="Event0" fromCardinality=""
   toType="G2of3" toCardinality="3"/>
 </edgeType>
</formalism>
```


Figure 2: The XML definition of the FT Metaclass

A Metaclass defines the kind of nodes and edges that a model may include, e.g. types of events, gates and arcs in Fig. 2. Nodes, edges, and formalisms themselves are all called *elements*.

Edges have also associated a set of *constraints* that tell which kind of elements that edge may connect. For example, the "Arc" element defined in Fig. 2 specifies which connections are allowed between events and gates. Constraints can also specify a *cardinality*: that is the maximum number of edges of that kind that may start "from" or end "to" a particular element.

Since constraints are expressed in terms of elements, an edge can connect not only two nodes, but also other edges and sub-models. As a result, Model Classes or Model Objects can be handled as they were nodes.

All elements have one or more "properties" that are the **private attributes** of the Model Classes and that will be set when creating a Model Object. An additional attribute called *visibility* is used to define the interface elements: the edges can connect elements according to their constraints, and also the elements of sub-models that have the visibility property set to true.

Turning to our example, an extension of the FT Metaclass with a Repair Event (RE) is necessary to extend the FT formalism and analysis techniques by adding repair actions. In Fig. 3 the RFT Metaclass is shown that inherits from FT and extends it by adding a "Repair" node and a proper edge so that the "Repair" node can be linked to one of the events in the tree. The "Repair" node is an implicit node: it relies on an external specification of the repair policy.

A "Repair" node corresponds to a RB (Repair Block), and since several repair policies

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE formalism SYSTEM "formalism.dtd">
<formalism parent="FT.xml" name="RFT">

<nodeType parent="BasicEvent" name="RepBE">
  <propertyType name="RepairDistribution" default="EXP 1.0"/>
</nodeType>

<nodeType parent="" name="RepairNode">
  <propertyType name="Name" default=""/>
  <propertyType name="RepairDistribution" default="EXP 1.0"/>
  <propertyType name="Policy" default="SingleRepairTime"/>
</nodeType>

<edgeType parent="" name="RepairArc">
   <propertyType name="RepLabel" default=""/>
   <propertyType name="EventLabel" default=""/>
   <constraint fromType="RepairNode" fromCardinality="1"
   toType="Event" toCardinality="1"/>
</edgeType>
</formalism>
```

Figure 3: The RFT Metaclass derived from FT

are possible, it must include some information on the particular policy it represents. In this paper we assume that a repair block causes the elimination of the fault event by eliminating all its potential causes. The properties "Policy" and "RepairDistribution" will be used to define the implicit behavior at the solution time.

Finally, the RFT Metaclass redefines the "BasicEvent" node ('RepBE' in Fig. 3) by adding a new property "RepairDistribution" used to specify the time distribution of the repair action needed when that BE occurs.

3.2 Model Classes and Model Objects in DrawNET++

Classes are useful to create a library of sub-models to be used by an end user. Fig. 4 shows a FT flat model representing a subsystem of the second version of the multiprocessor system (with no shared memory). A FT sub-model is a subtree whose interface is defined to be the top event of the subtree.

Once the FT flat model of the subsystem has been created, it can be saved as a Model Class, since it represents an abstraction of a system component. The final model of the multiprocessor system will be a composed model containing three instances of this Model Class, i.e. the three Model Objects $SUB_IND1, SUB_IND2, SUB_IND3$ graphically represented by squares in Fig. 5 (a). They are obtained by specifying different names for each object and giving distinct values to the properties of the elements of the Model Class in Fig. 4, for example the proper values of the fault rates of the BEs. The model in Fig. 5 (a) is a weak aggregation (i.e., a model obtained by instantiating and connecting submodels). Fig. 5 (b) shows a high level representation of the system obtained after applying strong aggregation (i.e. transforming the model in Fig. 5 (a) into a new submodel which hides the interfaces of the three submodels composing it). The interfaces (events SUB_i, $i \in \{1,2,3\}$)

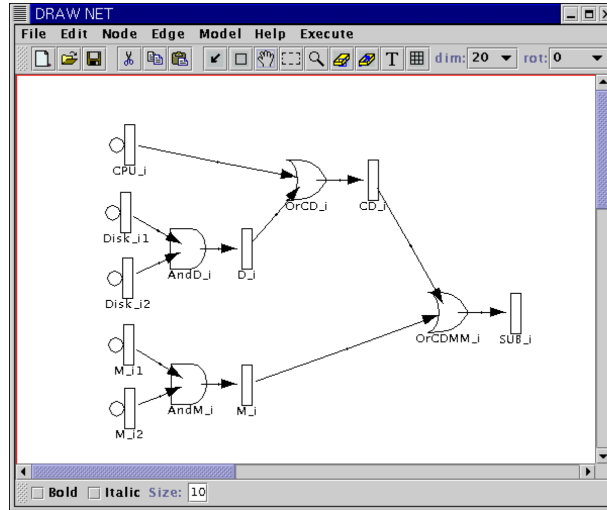


Figure 4: An example of Model Class

used to connect the three Model Objects to the G2of3 gate (in Fig. 5 (a)) have been hidden and they are no longer visible. The whole processor subsystem has been encapsulated in a Model Class whose instance named "processing" is used to build the complete model.

4 GSPN representation of RFTs

This section introduces the basic concepts needed to describe the multi-solution method of Sec. 5. In particular it is explained how an RFT component can be transformed into a GSPN model by (1) automatic translation of FT objects and (2) composition with the GSPN implicitly defined by each RB.

Automatic translation of a FT into a GSPN. Let us briefly explain the FT to GSPN translation algorithm: for more details the reader can refer to [11, 4]. Each Basic Event BE in the FT is modeled with the subnet in Fig. 6(a): the firing time associated with the timed transition represents the time to failure of that BE. Each gate in the FT is translated into one or more transitions, connected to the places representing the input/output events of the gate (see Fig. 6(b) and (c)).

The subnet representing all BEs, and those representing the gates, are then superposed on places with equal label, forming the logic structure of the FT. An example is given in Fig. 6(d), where the translation of the subtree of the multiprocessor FT starting at event CD1 is shown. The state space of the GSPN represents all possible evolutions of the model through its possible failure states.

GSPN models of repair boxes. Let us introduce some possible semantics for the RBs and their translation in terms of a GSPN that can be automatically composed (through

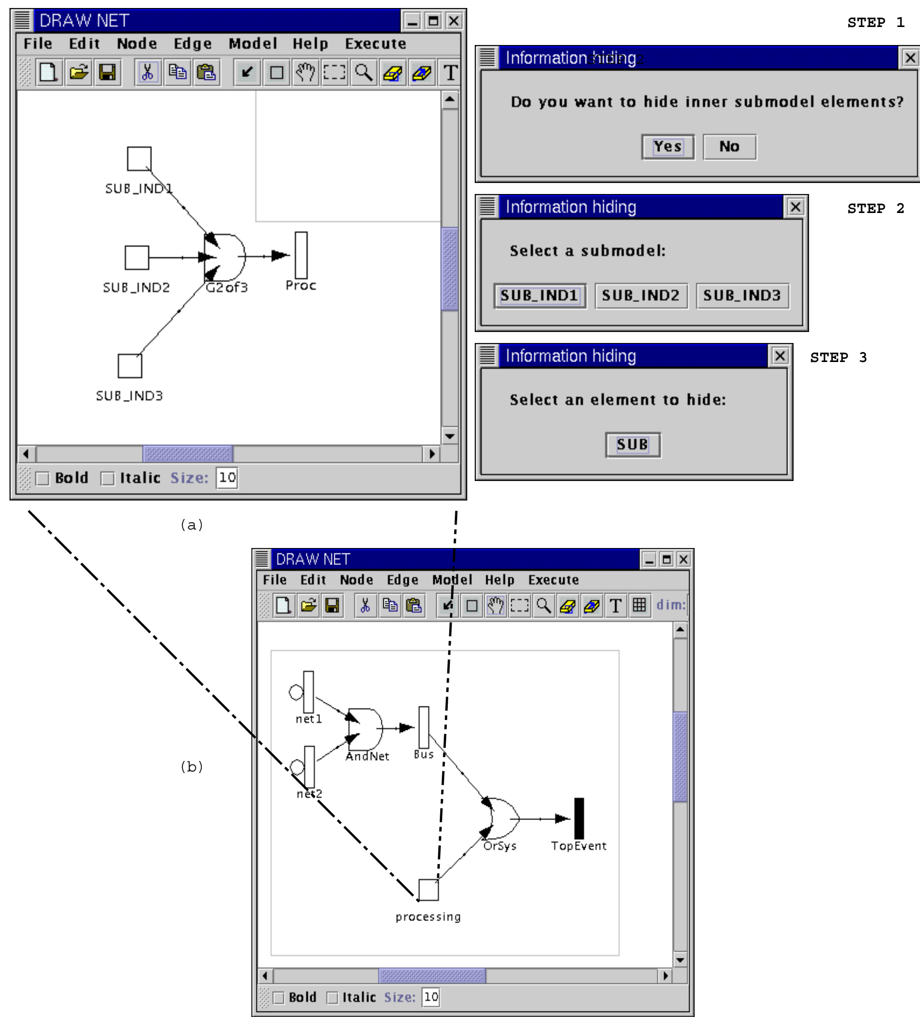


Figure 5: An example of weak and strong aggregation: the FT model of (a) the processor subsystem and (b) the multiprocessor system

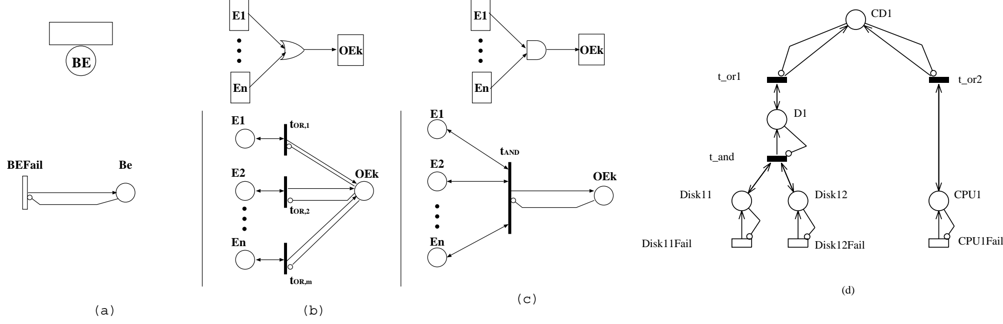


Figure 6: FT to GSPN translation rules: (a) BE (b) OR (c) AND (d) a subtree

place and transition superposition) with the GSPN of the FT (generated as explained in the previous subsection).

Let OE_1 be a repairable event of an FT (i.e. an event connected with an RB): it identifies a subtree of all events that may lead to the occurrence of OE_1. Let us assume that OE_1 represents the failure of a given system component C: depending on the type of tree originating in OE_1, the repair actions allowing to bring component C back into the operational state may differ. Hence the RB should include enough information to express the repair strategy to be followed: in the rest of this section we shall consider a repair strategy called *complete repair*, consisting of repairing all the basic subcomponents corresponding to the BE leaves of the subtree originating in OE_1 (which has the side effect of repairing all the events on the path from the leaves to OE_1). Other strategies are conceivable, and in principle any strategy that may be modeled by a GSPN might be directly embedded in the RB node by explicitly associating a GSPN submodel to it. In this section we show how the GSPN of the *complete repair* strategy can be described in parametric form and automatically generated.

Let us consider the multiprocessor FT example in Fig. 1. If a repair box is connected to event $CD1$, the complete repair strategy would require to repair the BE $Disk11$, $Disk12$ and $CPU1$ (and hence indirectly also $D1$). Observe that repairing $CD1$ may affect also the fault status of SUB_1, $Proc$ and the TE.

There are some possible choices in modeling the *complete repair* strategy:

(1) the time required to perform the whole repair may be modeled by a single timed transition or as a (parallel or sequential) composition of several times, one for each occurred BE in the subtree; in the former case, this time should depend on which subset of BE has caused failure event OE_1;

(2) possibility of new BE occurrence during the repair of OE_1: when component C is under repair, is it possible that some not yet failed BE fails during the repair? And once a subcomponent has been repaired, may it fail again before the whole repair of C has completed? (this last case of course can be considered only if the OE_1 repair action is modeled as a set of separate repair actions, one for each basic subcomponent);

(3) infinite versus finite repair facilities: it might be necessary to consider the fact that a repair action can take place only if a repair facility is available to perform it. Repair facilities may be local to a RB, or shared by several RBs. If at most n repair facilities are available to a given RB, the time needed to complete the repair will have to be adjusted

accordingly. Moreover if the repair facilities are shared, we may want to establish a policy for the assignment of such facilities to RBs (e.g. a policy based on priorities, with or without preemption).

Let us sketch two parametric GSPNs modeling the repair of a given component, both assuming that new BEs may occur during the repair, and differing in the number of timed transitions used to represent the whole repair. In case several timed transitions are used to model the repair we assume that an already repaired BE cannot occur again before the whole component C repair has completed. For the sake of space we make a simplifying assumption: the subtree originating in OE_1 does not contain any *shared* BE (in other words, any path from the TE to the leaves of OE_1, contain OE_1). Observe that shared BEs may exist since an FT may actually be a DAG instead of a tree (as is in the case of the running example when a unique shared memory is used for replacing a failed local memory).

The two GSPN models of a repair box for $CD1$ are depicted in Fig. 7 and Fig. 8: the subtree originating in $CD1$ includes the event $D1$ and basic events $Disk11, Disk12$ and $CPU1$. Moreover, on the path from the TE to $CD1$ there are two events, namely SUB_1 and $Proc$. The two models can be easily generalized to an arbitrary number of BEs and intermediate events (by repeating the same pattern for each BE and event in the subtree). The first model comprises an immediate transition and a place (StartRepCD1 and RepCD1) representing the start of repair, a timed transition RepTimeCD1 modeling the time needed to complete the repair and a subnet for the deletion of the tokens representing a failure from all places corresponding to events in $CD1$ subtree, as well as from the places corresponding to the TE and the events on the path from the TE to $CD1$ (in the example SUB_1 and $Proc$). By so doing, not only all events potentially causing $CD1$ are cleared upon repair, but the effect of the repair is also propagated to all the events which depend on $CD1$, all the way up to the TE: in case the repair of $CD1$ is not sufficient to make the upper events operational, then the GSPN shall automatically regenerate the appropriate tokens in the corresponding places using the immediate transitions representing the FT gates logic (transitions t_or1, t_or2 and t_and in the example of Fig. 6(d)).

The second model is slightly more complex, since several timed transitions are included ($RepDisk11$, $RepDisk12$ and $RepCPU1$), representing the repair time of a single basic subcomponent. The enabling of this transitions is conditioned on the fact that the corresponding failure has occurred (e.g. input place CPU1_ko) and that the repair action has started (input place RepCD1). When the repair has completed, place CPU1_rep becomes marked, place CPU1_ko is emptied, while place CPU1 remains marked (preventing the occurrence of further failures for CPU1). The same holds for each basic event BEi in the subtree. When all BEs are OK (places BEi_ko all empty) the repair has finished, and all failure tokens can be removed (by the same subnet of high priority immediate transitions already explained for the GSPN in Fig. 7). Observe that in this model we are assuming that there are at least as many repair facilities as the number of BEs in the subtree: in fact, the repair actions of all BEs can proceed in parallel.

Other variants of these models are possible, for example for handling shared BEs, for modeling limited repair facilities, or for forcing a given order in the repair of the BE in a subtree: for space reason they are not presented in this paper.

The repair submodel(s) can be composed with the GSPN representation of the FT by applying a composition operator which glues together two models by superposing places or transitions with same label in the two nets: in our case the models should be superposed over the places representing events, and on the transitions BEiFail, representing the

A symbol π_i next to an immediate transition t means that t has priority i.

Figure 7: Sketch of repair net with only one repair time

Figure 8: Sketch of repair net with several repair times.

occurrence of basic failure event BEi. Once the complete GSPN has been obtained, the unavailability probability of the TE at a given time t can be computed (e.g. using the transient analysis module of the GreatSPN package [7]).

5 Solving the model

In this section a strategy is presented to solve the composed models described in the previous sections. The logical architecture of the interface to the involved solvers is outlined. Indeed, our methodology requires that solving strategies and their related solution procedures are defined in order to achieve multi-solution analysis of multi-formalism models.

5.1 A multi-solution strategy

A central issue in approaching complex FT models analysis is choosing the most efficient solution method. State space based solution techniques can be very computation intensive: on the other hand the efficiency of combinatorial methods can be exploited only at the price of giving up advanced features such as repair. For this reason a modular approach is proposed in this paper to take advantage of both techniques.

Our method is based on the identification of the set of independent modules to be solved with different techniques.

We introduce the following definitions:

Definition 1 *An independent subtree without repair, also called* **combinatorial solution module (CSM)***, is a subtree that (a) does not share any event with other subtrees, (b) does not contain any RB, (c) none of its ancestors is connected to an RB. A CSM is maximal if it is not contained in another CSM.*

Definition 2 *An independent repairable subtree, also called* **state space solution module, (SSM)***, is a subtree of the whole FT that (a) does not share any event with other subtrees, (b) contains at least one RB, (c) none of its ancestors is connected to an RB. A SSM is minimal if it does not contain neither a CSM nor a (smaller) SSM.*

Definition 3 *Let F be a FT including a SSM or CSM module $\mathcal{M}$. A $\mathcal{M}$ –* **reduced** *FT $F^R_{\mathcal{M}}$ can be obtained from F by substituting $\mathcal{M}$ by a Basic Event $BE_{\mathcal{M}}$ "equivalent" to $\mathcal{M}$. A Basic Event $BE_{\mathcal{M}}$ is said to be* **equivalent** *to a module $\mathcal{M}$ if its occurrence probability at a given time $\bar{t}$ is equal to the occurrence probability of the Top Event of $\mathcal{M}$ at time $\bar{t}$.*

The basic idea of the multi-solution strategy consists of iteratively reducing the whole FT by substituting independent subtrees with an equivalent BE whose occurrence probability is obtained by solving the corresponding subtree.

Observe that it may be the case that the combinatorial techniques apply some form of decomposition for efficiency reasons: we do not deal with this aspect in this paper.

Fig. 9 shows an example of reduction applied to the multiprocessor system with no shared memory and with repairable subsystems SUB_i (i.e., we are assuming that a RB has been connected to each event SUB_i). In this case the three SUB_i subsystems are SSM

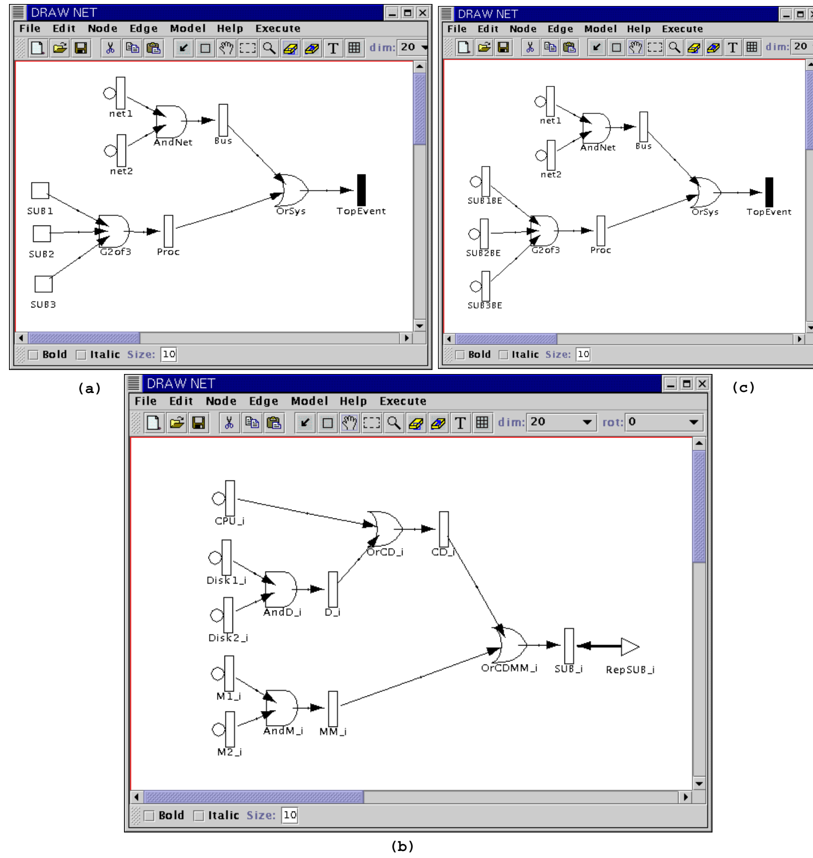


Figure 9: FT decomposition: independent subtrees. (a) the complete FT (b) repairable subsystems (SSM modules) (c) the reduced FT after SSMs substitution

modules. Each one is translated into a GSPN net and solved by a proper analysis tool, e.g. GreatSPN, as explained in Sec. 4.

The resulting fault probability is the occurrence probability of the equivalent BE "SUB$_i$BE". The resulting FT model can be solved by combinatorial analysis techniques, e.g. by means of the SHARPE package [18].

Fig. 10 shows an example of reduction applied to the multiprocessor system with shared memory, assuming that event MM_i is repairable (Fig. 10(a)). The presence of M_s introduces a dependence in the system. The reduction can be initially performed on the three CSM modules consisting of the CPU_i and the local disks (CD_i subtree). Each of them is solved by the SHARPE tool. The resulting fault probabilities are the occurrence probability of the equivalent BEs "CD_i" in Fig. 10(a). Then the whole processing subtree is translated into a GSPN and solved by GreatSPN. The obtained fault occurrence is assigned to the equivalent BE "processing" (see Fig. 10) and the resulting FT is solved by the SHARPE tool.

5.2 The logical architecture of the interface to the solvers

Fig. 11 describes the DrawNET++ interface to the solvers. In this case we suppose that the analysis tools to be used in the multi-solution technique are the GreatSPN and SHARPE packages.

The architecture consists of two levels:

1. The RFT hierarchical pre-processor level, that is responsible for analyzing the XML representation of the model produced by DrawNET++, identifying and separating the XML parts describing SSM and CSM modules. The RFT hierarchical pre-processor has also to manage the RFT structural information to insert the failure probability of the equivalent BEs into the higher level modules in the hierarchy.

2. The Translator level, that is responsible for the processing of the XML description of the SSM and CSM modules. In particular the tools at this level have to: a) generate the GSPN translation of the Repair Blocks (RFT2PN); b) generate a GSPN model representing the FT component of the RFT module (FT2PN); c) generate the proper XML format versus external tools by means of XSL filters (FT2SHARPE).

The GSPN net produced by the RFT2PN and FT2PN module are then composed with the ALGEBRA tool [3] which returns the complete GSPN RFT module.

Where XML is not employable in the translation versus external solvers, due to the need for proprietary formats, an appropriate translation is performed to/from XML to normalize parameter exchange.

Finally, global results are translated in XML and returned to DrawNET++ so that they can be shown to the user by the graphical interface.

6 Conclusions

In this paper we have shown the feasibility of a multi-formalism multi-solution approach based on the definition of a simple language (Metaformalism) to define graph formalisms

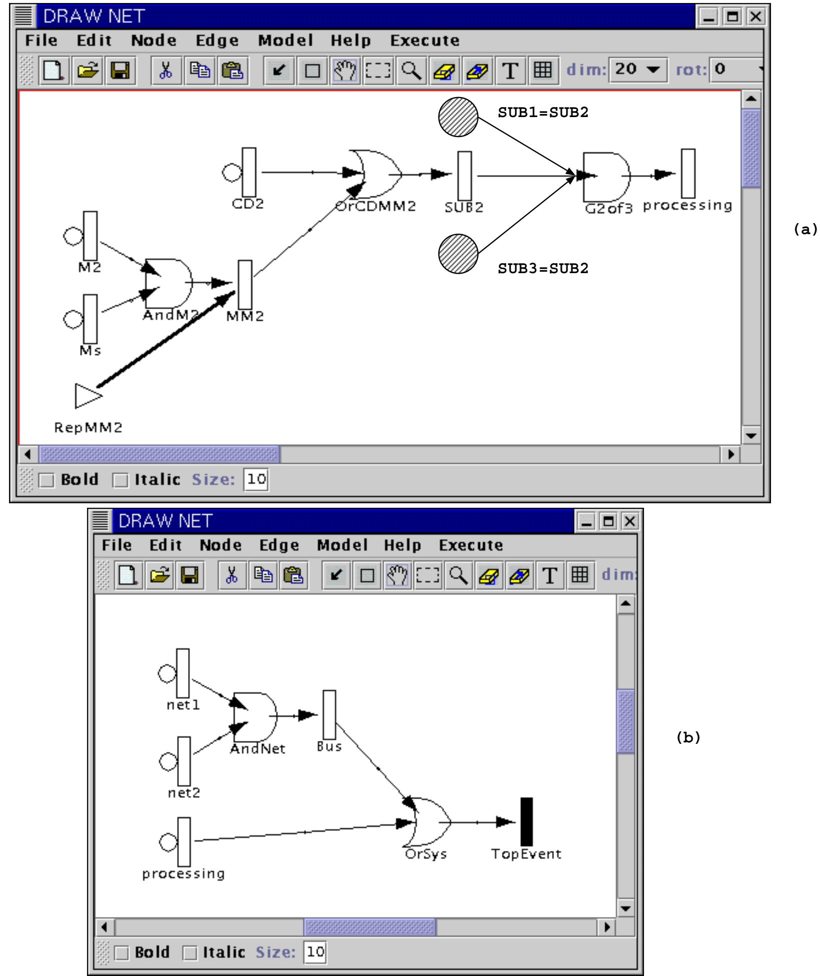


Figure 10: FT decomposition: (a) a SSM module including several dependent subtrees (SUB_i) (b) the reduced FT after SSM substitution

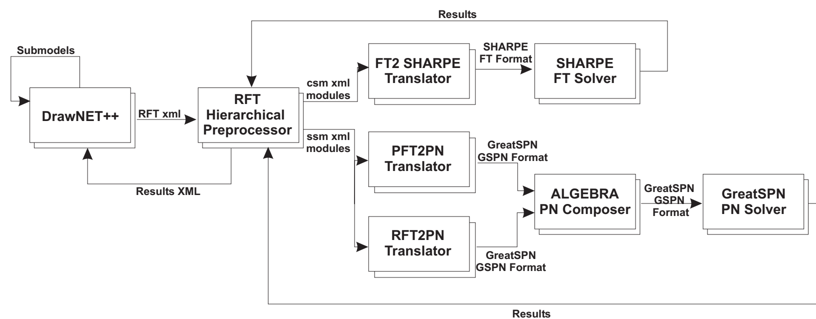


Figure 11: The interface to SHARPE and GreatSPN

and on a OO methodology to develop multiformalism models. Multi-solution is achieved by defining a proper back-end interface versus different solvers. The overall approach is partially supported by the DrawNET++ framework. With respect to other approaches, we allow to easily design new formalisms and models structuring and composition schemes, as well as support to interoperability of different solvers.

The application example used throughout the paper concerns the construction and analysis of dependability models using two different formalisms, namely FTs and GSPNs. The proposed method however is by no means limited to this application domain nor to these formalisms, on the contrary the flexibility of the approach allows to apply it to any application area and model description formalism and related solution methods and tools.

The dependability example considered in the paper is based on a user level formalism appearing as a simple extension to FT, called RFT, including Repair Blocks (RB); both formalisms can be easily defined within DrawNET++ using its Metaformalism language. Several different repair strategies can be defined and embedded into the RB nodes, in the form of a parametric GSPN model representation, with an interface suitable for composition with the GSPN translation of the repairable subtrees in the FT.

Models can be structured in a hierarchy of submodels, allowing to both manage the complexity of the model design process, and also partition the model in such a way that it suggests how a multi-solution algorithm may be applied (with the support of DrawNET++). An architecture has been defined for a DrawNET++ back-end that constitutes an interface between DrawNET++ and the SHARPE FT solver and the GreatSPN GSPN solver. The basic idea consists of repeatedly applying the most appropriate solution algorithm to the FT submodels (starting from the inner ones in the hierarchy), and bringing the result back into the upper level models. An additional module could be integrated in the post processor, implementing an algorithm to check the proposed hierarchical structure, and possibly modify it (in a user transparent way) for achieving a more efficient combination of solution techniques.

In the proposed example the hierarchical nature of the model suggests a simple hierarchical solution scheme, however any (more complex) scheme might be embedded in the back end, based on the assumption that the model structure (implicitly or explicitly) includes all the needed information to decide how to decompose it for applying the multi-solution algorithm, and that there is a way to make the solvers interact (typically by feeding the results from one solver into the input model of another solver). Of course the implementation of the back end realizing the connection between different formalisms and solvers is not trivial, however we believe that this approach is appealing because it enables the reuse of existing solvers, and allows to quickly experiment in a flexible way combinations of different solvers.

References

[1] A. Anand and A.K. Somani. Hierarchical analysis of fault trees with dependencies, using decomposition, In *Proc. IEEE Annual Reliability and Maintainability Symposium, pp. 69-75*, 1998.

[2] P.Ballarini, S.Donatelli and G.Franceschinis. Parametric Stochastic Well-Formed Nets and Compositional Modelling, In *Proc. 21st International Conference on Application and Theory of Petri Nets*, Aarhus, Denmark, June 26-30, 2000.

[3] Bernardi S., Donatelli S., Horváth A., Special section on the practical use of high-level Petri Nets: Implementing Compositionality for Stochastic Petri Nets. *Int. Journal of Software Tools for Technology Transfer (STTT)*, Springer Verlag ed. - vol.3 issue 4, August 2001, pages 417–430.

[4] A. Bobbio, G. Franceschinis, R. Gaeta and G. Portinale. Dependability Assessment of an Industrial Programmable Logic Controller via Parametric Fault-Tree and High Level Petri Net, In *Proc. 9th International Workshop on Petri Nets and Performance Models*, Aachen, Germany, Sept. 2001.

[5] G. Booch. ObjectOriented Design, BenjaminCummings, Redwood City, CA, 1991.

[6] R.M.L.R. Carmo, L.R. de Carvalho, E. de Souza e Silva, M.C. Diniz and R.R. Muntz. Performance/availability modeling with the Tangram-II modeling environment, Performance Evaluation 33 (1998), pp.45-46.

[7] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets, Performance Evaluation, vol. 24, no.1-2, Nov. 1995, 1995, pp.47-68. http://www.di.unito.it/∼greatspn/

[8] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed coloured nets for symmetric modelling applications, *IEEE TOC*, vol.42 pp.1343-1360, 1993.

[9] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. Doyle, W. Sanders, P. Webster. The Möbius modeling tool, In *Proc. 9th Int. Workshop on Petri Nets and Performance Models*, Aachen, Germany, Sept. 2001, pp. 241-250.

[10] D. Deavours and W. Sanders. Möbius: Framework and Atomic Models, In *Proc. 9th International Workshop on Petri Nets and Performance Models*, Aachen, Germany, September 2001.

[11] R. Gaeta, A. Bobbio, G. Franceschinis and G. Portinale. Exploiting Petri Nets to support Fault Tree based dependability analysis, In *Proc. 8th International Workshop on Petri Nets and Performance Models*, Zaragoza, Spain, September 1999.

[12] Y. Dutuit and A. Rauzy. A linear-time algorithm to find modules of fault trees, IEEE Transactions on Reliability, Vol. 45, No. 3, September 1996.

[13] G. Franceschinis, M. Gribaudo, M. Iacono, N. Mazzocca and V. Vittorini. DrawNET++: Model Objects to Support Performance Analysis and Simulation of Complex Systems, In *Proc. 12th International Conference on Modelling Tools*

and Techniques for Computer and Communication System Performance Evaluation (TOOLS 2002), London, UK, Lecture Notes in Computer Science, Volume 2324, Springer, pp. 233-238, April 2002.

[14] M. Gribaudo, A. Valente. Framework for Graph-based Formalisms, In *Proceeding of the first International Conference on Software Engineering Applied to Networking and Parallel Distributed Computing 2000, SNPD'00*, pages 233-236 Reims, France, May 2000, ACIS.

[15] H. Henley and H. Kukamoto. Reliability Engineering and Risk Assessment, Prentice-Hall, Englewood Cliffs, 1981.

[16] N. Leveson. Safeware: System Safety and Computers, Addison-Wesley, 1995.

[17] K. Jensen. Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Vol. 1: Basic Concepts, EATCS Monographs on Theoretical Computer Science. Berlin: Springer-Verlag, 1992.

[18] R.A. Sahner, K.S. Trivedi, and A. Puliafito. Performance and Reliability Analysis of Computer Systems, An Example-Based Approach Using the SHARPE Software Package, *Kluwer Academic Publisher, November 1995, Boston,* 1995. http://sun195.iit.unict.it/GISharpe/index.html

Towards Automated Checking of Component-Oriented Enterprise Applications

Jukka Järvenpää Marko Mäkelä*
Laboratory for Theoretical Computer Science,
Helsinki University of Technology,
P.O.Box 9205, 02015 HUT, Finland

July 29, 2002

Abstract

Building enterprise applications using component-based frameworks has been suggested as a way to help companies manage their software assets. We propose tool support for managing these high-level data-centric applications with formal methods. Our method is based on extracting a system model from the models of components and from the application code which glues the components together. This model is used for generating state spaces that can be checked for desired or undesired properties. In order to manage the state space explosion problem we propose that the application developer controls some parameters of the model. Even though the insight of the application developer is still needed, we believe that creating tool support for the proposed method could contribute to the success of the component-based approach.

Keywords. software components, transactions, abstractions, verification, Java

1 Introduction

Enterprise application systems have traditionally been used to integrate internal business processes within companies.

The current trend is to expand integration across organisations. The objective is to create more dynamic trading partner relationships, to reduce costs and to increase the productivity of companies participating in a networked economy. This trend sets high demands for companies to maintain and modify their core systems.

Enterprise application systems have often evolved from in-house development projects. The alternative is to buy a packaged solution from an outside software vendor. Compared to a packaged product, an internally developed system could better match the needs of the company. On the other hand, in-house development costs must be carried solely by the company, while software vendors can distribute their costs to a larger number of clients. Also, the package vendor gains experience

*This research was financed by the Helsinki Graduate School on Computer Science and Engineering, by Jenny and Antti Wihuri Fund, and by Academy of Finland (Project 47754).

when delivering solutions to different companies, which allows it to incorporate best practice into the package. With internal development this is harder to achieve.

Buying a packaged solution does not come without difficulties either. When the package is installed and configured, the final result can be more determined by the abilities and options of the package rather than the needs of the organisation [12]. Choosing a monolithic package is a commitment that locks the customer into a business relationship with the vendor for a long time. Sometimes this is mutually beneficial, but it could turn out to become harmful if the vendor is not capable of offering the support needed, or goes out of business.

1.1 The Component Approach

A middle course between the "make" and "buy" approaches is to build the system from reusable components. In this approach, the core system contains only minimal functionality, and the necessary tailoring is done by composing distinct encapsulated entities within the system framework. Component-based frameworks are partial implementations that provide fundamental elements, structural integrity and extension points.

Components are packaged software artifacts that provide functionality through a set of well defined interfaces. Component-based systems are expected to become a key business productivity solution for suppliers and consumers in the application market [22]. The anticipated benefit is a flexible and economical infrastructure, where organisations have a considerable choice of procurement to create customised solutions [22]. System acquisition and modifications should also become more manageable, because the modular architecture allows components to be deployed and updated individually [12]. Well-defined interfaces isolate component development from the rest of the system.

Figure 1 gives a simplified picture of a component-based framework application. The picture demonstrates how the framework invokes application code, which extends and refines the framework. The application code acts as glue between the framework and the components. Some of the business rules are contained within the application code, but most program code, such as database access, is hidden behind the component interfaces.

We believe that there is a great demand for tool support for managing applications built using component-based frameworks. This article presents a proposal for extracting models from application code and the components it accesses. These formal models can be explored to check whether the application behaves as required. Many problems must be solved to make such an approach possible. Among other things, the process of extracting a model should be highly automated, and the state spaces generated by the resulting model should at the same time be both manageable and correspond to the implemented behaviour. Last but not least, application coders must be able to specify the system requirements and to see the error traces in terms familiar to them.

1.2 Outline

The rest of this article is organised as follows. Section 2 discusses the economic and environmental preconditions that must be satisfied before software verification can be used for managing component-based framework applications. Section 3 describes an application environment in terms of architecture, software processes and tools that make it possible to extract verifiable models from applications developed in the environment. It also contains a code excerpt from a sample

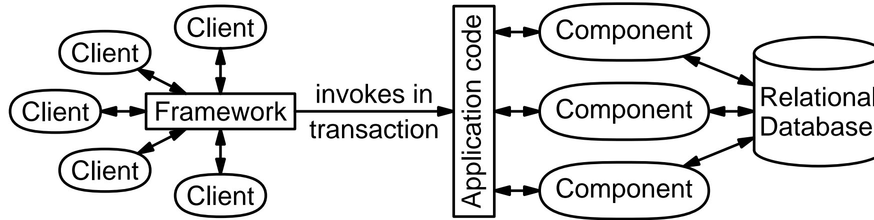


Figure 1: Simplified view of an application in a component-based framework.

application we use to clarify our method. Section 4 defines our modelling framework of enterprise applications at a conceptual level. It describes what kind of questions the model should be able to answer, and it discusses some modelling considerations, which must be taken into account. Section 5 revisits our example and shows how a developer might use the proposed tool. Finally, we discuss some related work and conclude our presentation.

2 Preconditions for Component Software Verification

2.1 Economic and Environmental Preconditions

The component approach, as such, does not guarantee to solve all the problems in enterprise application software management. The components and the framework must be designed to meet industry requirements—not a trivial task at all. Everything must adapt to the customer environment and be manageable by both the customer and the vendor. The integrity and the functionality of the system must be guaranteed even when third party components are integrated. Conventional software engineering practices, such as requirements analysis, system modelling, version control, testing and documentation, retain their importance in component-based system development.

More advanced software engineering techniques, such as automated software verification, could contribute to the success of component-based systems. Applying formal methods to component systems gives a profoundly different starting point for third-party component markets. A formal model—an abstract description of a system—can be thoroughly analysed by computer tools to increase confidence in the system working according to the specification. System models can be derived by composing the high-level application logic with models of the system framework and the components. Verification techniques have the potential to decrease maintenance costs, too. Costs could be saved by simulating or verifying the impact of application changes on a formal model. Automated verification runs could replace some of the otherwise required testing.

2.2 Architecture, Process and Tool Preconditions

To successfully apply verification techniques in industrial-scale application development, the environment has to fulfil a number of requirements:

Precisely defined architecture. As verification is based on a model, the results are meaningful only if the model corresponds to the executable application on an abstract level. This requires that the application structure is precisely defined and implemented.

High quality repeatable processes. In the same way, the correspondence between the model and the application necessitates that the design processes used to create the executable code and the verifiable model are repeatable and of such quality that small deviations in the design process do not lead to substantial differences.

Integrated tool support in the development environment. Constructing verifiable models manually would consume too much time and require highly specialised skill. Automated tool support eliminates these problems as well as errors in translation. Verification tools should accept input directly from the elements created by the developer in the design domain and map the output back to the design domain.

3 The Environment of the Component Framework

3.1 Enterprise Application Architecture

Enterprise applications are data-centric systems where persistent data is stored in databases and processed by application programs. Typically, enterprise applications build upon a client–server architecture where business rules are implemented on the server side, and clients take care of the user interface.

In industrial software packages, databases usually follow the relational model [23, Chapter 2.3]. The conflicts that may arise when several processes access the database simultaneously are resolved using *transactions* [23, Chapter 9]—atomic sequences of operations. Either the effect of all operations are committed to the database, or the whole transaction is rejected.

In a database management system, operations belonging to different transactions are interleaved with each other for performance reasons. In a formal model, the operations of database management system can be abstracted by serialising the transactions, allowing the model to process only one transaction at a time.

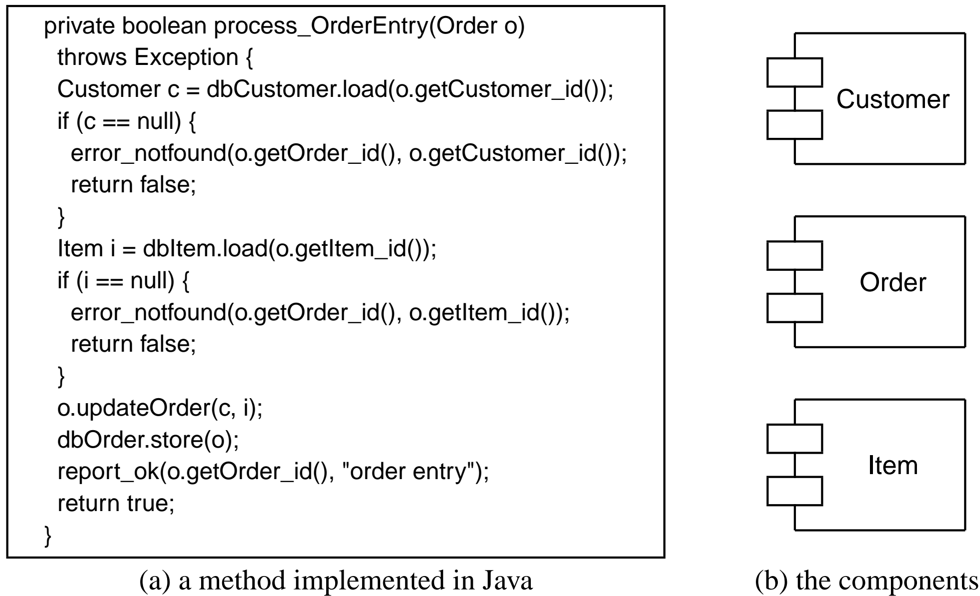
3.2 An Example Application: Processing Orders

To gain more insight into component-based enterprise application frameworks, we show an extract from an example application in Figure 2(a). The application code[1] is invoked by the framework when an order is entered. The involved components are shown in the UML diagram of Figure 2(b). The code retrieves customer and item information from the database, updates the order with this information and stores the order into the database.

The semantics of the example deserves some additional remarks:

- If the method raises an exception or returns the error code of `false`, the framework will roll back the transaction, so that no changes are committed to the database.
- The method does not store any internal state between successive calls. The persistent state is kept in the database.
- Most of the implementation is hidden behind component interfaces.

[1] This method could be implemented in the J2EE architecture [21] in a session bean.



(a) a method implemented in Java

(b) the components

Figure 2: An application for processing orders.

  - The objects dbCustomer and dbItem are simple components, whose load methods simply retrieve objects from the database.
  - The composite component dbOrder hides a more complicated implementation. In this example, we assume that the method store tries to combine the new order with an open order the customer might have. If no such order exists, a new order is stored into the database.

- Some code, such as calls to the logging facilities error_notfound and report_ok, does not affect the state of the application and should be omitted from the model.

3.3 Software Processes

Maintaining a component-based software system requires that repeatable processes be followed to manage the framework, the components, and the application code. The majority of the application lifetime costs are incurred by the maintenance period [20, Chapter 30]. From the customers' point of view, most maintenance tasks are likely to concern application code modifications and occasional deployment of new components.

In order to make application modifications more effective, we propose that automated verification takes place before system level testing. The objective is to gain more insight into the application than could be achieved by pure static analysis techniques. In this step, a system model—derived from the application code and the components—is explored with a verification tool that presents any errors as executions of the application code.

The proposed automated verification step requires that for each deployed component, there is a model of its implementation. In order to guarantee this, both the models and the implementations should be the results of the component design process.

The tools that assist in these processes are described in the following section.

3.4 Tool support

To automate the verification step, we need a tool which parses the application code, accesses a library of component models, composes the parsed application code with the component models and feeds the result to a model checker. This tool should also map any error traces from the model checker back to execution traces of the application code.

To ensure that a component implementation conforms to its model, we propose the following procedures to be aided by tools:

Automated derivation of simple models. Models for simple components could be produced automatically from the same repository information from which the implementations are generated. Examples of such components are object/relational mapping routines, which allow the data in relational databases to be stored and retrieved as objects.

Automated derivation of composite models. When a component is implemented by wrapping other components together with application code, its model can be derived automatically by composing the parsed application code with the models of the wrapped components. This can be accomplished with the same tool that creates system models.

Manually maintained models. Models for the most complex components must be maintained manually. This is tedious, but the involved cost is justified if the component can be sold to several installation sites.

Manual work easily leads to differences between the model and the implementation. Conformance testing [8] could help to locate the errors. The manually constructed component model acts as the specification that the implementation can be formally tested against. Again, conformance testing should be supported by tools.

4 Formalising Component-Based Applications

In order to analyse a system, an automated tool needs a description of both the implemented and the desired behaviour. The system implementation is transformed into a formal model that generates a state space, such as a high-level Petri net. The desired properties are formulated in logic or as automata. Some properties can be derived automatically, others are retrieved from a library or specified by the application developer.

A model of an enterprise application is bound to have a huge number of reachable states. Therefore, the model must be structured and designed carefully. This section describes the main elements of the model and how they relate to the application. It also discusses the properties we would like to extract from the state space graph, and how the model should be built to limit the effects of the state space explosion as much as possible.

4.1 Modelling Elements

The core model can be mapped to shared memory multiprocessing. The shared memory is the database, and the competing processes are the transactions initiated by the environment.

These elements relate to the architecture in Figure 1 in the following way:

Environment. The environment models the application framework and the inputs from the clients. When state space exploration techniques are applied to a model, the model must represent a closed system, which means that the behaviour of the environment must be specified. The environment invokes methods of the application code, initiating a transaction for each request.

Transactions. Transactions model service execution within the application framework. If all operations succeed within the application code and within the components invoked to serve a request, the changes made to the persistent objects are committed to the database. Otherwise, the persistent state remains unchanged as the transaction is rolled back.

Application code. Application code may implement business logic or components, or extend or connect existing components.

To ease the extraction of models, application code is written in a subset of the Java programming language, comprising assignments, conditions, loops and virtual method calls. Some constructs, such as threads, have been deliberately excluded.

Simple components. These are the basic building blocks made available to application developers. Each operation in the component interface is defined with one or more transitions. The model can behave nondeterministically.

Database. The database is the persistent data-store of the application. Operations are grouped in transactions, which can be either committed or rolled back.

4.2 The Properties

Verification or model checking refers to the process of checking whether a model of a system behaves according to its specification. Automating this step requires that both the model and the behaviour requirements are in machine readable format.

Model checking is a useful tool in situations where new functionality is added to the system. Implementing the functionality might require changes to be made in several locations in the application code, and the application coder would like to gain assurance that he has correctly identified these locations.

When a property is violated, the verification tool should report an error trace, an execution sequence leading from the initial state of the system to the error. At the coarsest level, the error trace should display the names and parameters of the components that are executed. Sometimes the user would like to view parts of the trace in more detail, showing individual statements and variables in the application code.

In enterprise applications, many properties can be derived automatically from database definitions and program code. Only high-level requirements need to be formulated interactively.

4.2.1 Safety Properties

Safety properties are requirements on finite executions. Intuitively, they are statements of the form "nothing bad happens". For example, if a business function requires that a new database field is always initialised in certain business situations, the application coder can phrase rules or assertions such as "Field x is set whenever y holds."

Enterprise application databases are most likely designed to contain fields recording status information, such as whether an order has been accepted, or whether it has resulted in a delivery or a sent invoice. Safety properties can express requirements which may refer not only to several such status fields at once, but also to a history of states. This allows us, for example, to verify that the status fields fulfil a requirement such as "if an invoice is sent, a delivery must have occurred and the order must have been accepted."

Data integrity rules. Many relational database management systems have built-in mechanisms for ensuring the integrity of stored data. It is possible to restrict the set of allowed tuples by defining row constraints (e.g., "the delivery date of an order must be either null or later than the registration date") or foreign keys (e.g., "each order item row must refer to an existing order").

Whenever a tuple is inserted, modified or removed, the database management system checks all relevant rules and rolls back the transaction if any rule is violated. The rules form a safety net against errors that may occur in exceptional situations. These rules might never be violated in basic tests, but exhaustive verification will find all violations by testing all possible cases.

Assertions in program code. Many programming frameworks include an assertion facility. The program code may be instrumented with Boolean conditions that reflect the programmer's assumptions. Rules can be specified for the data passed to or returned by methods, or as arguments to a special "assert" macro that aborts program execution if the specified condition does not hold.

Such assertions can be automatically transformed to safety properties of the model. Similarly to database integrity rules, the assertions are most likely to fail in exceptional situations that can be best found in exhaustive testing.

Identifier pool alert. Section 4.3.2 explains why abstract identifiers are needed in the model and describes our solution for managing the state space explosion problem by using small enough data domains. Deadlocks may occur if these identifiers run out. This is not necessarily an error in the application, but it may be caused by the model where the number of available identifiers is limited. A safety guard can assist the user in managing the identifier domain sizes. When the last identifier is taken, the safety guard is triggered to indicate a potential problem. The occurrence of this event would suggest that the domain should be enlarged. Such checks should be optional.

4.2.2 Liveness Properties

Verifying that a system never reaches an erroneous state is a very powerful way to increase confidence in the correctness of the system. However, sometimes this is not enough, and we want to claim that "something good eventually happens," such as "an order entered into the system will eventually also be processed."

A liveness property is violated if there is an infinite execution where progress is not guaranteed. Usually this means that some actions can be repeated infinitely in the system, and the same states are visited again and again.

When expressing liveness properties we need also to assume that certain actions receive fair treatment. When strong fairness is assumed for a transition, it must be executed infinitely often if it becomes enabled infinitely often.

In our example, where orders are entered and processed separately, we must assume that neither transaction is neglected in order to verify that each order eventually results in a delivery.

Some of the more complicated application behaviour requirements can only be specified by the designer, who expects the application to behave in a certain way. This task can be eased by providing the designer with *specification patterns* [6], templates of formulae or property automata.

4.3 Modelling Considerations

We shall now consider the modelling elements from Section 4.1 in more detail.

4.3.1 The Environment

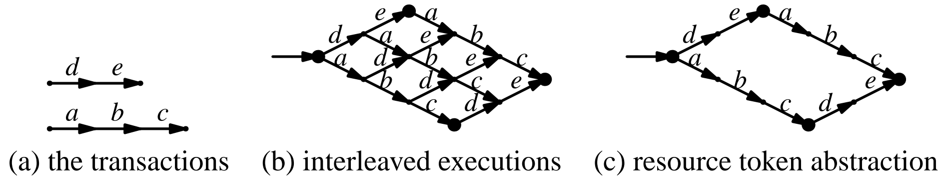
Domains of transaction parameters. The domains of transaction parameters greatly affect the number of reachable model states. Validated input is stored into a database, which can become quite large. This behaviour is reflected in the model so that enlarging the input domains result in even larger state spaces. In order to manage the state explosion problem, we have to limit the domains. When the application developer is allowed to select the input domain sizes individually, he can check different aspects of the system. Obviously this approach relies on the intelligence of the user and does not prove the absence of errors. However, checking a restricted model might reveal errors more easily than testing or simulating a more complete model.

Automatic unification of transaction parameter domains. The application code is statically analysed to identify the relations between database fields and transaction parameters. Each group of related fields and parameters is assigned an own domain. Developers cannot be assumed to keep such mappings up to date, as the system is maintained over a long period of time by different persons. Unifying the domains is essential for models with scalable domain sizes.

Controlling transaction invocations. One way to attack the state space explosion is to guide the search by restricting the behaviour of the environment. For instance, transactions for filling in basic information could have priority over the actual processing transactions. One way to arrange this is to divide the behaviour of the environment into phases where only certain transactions will be invoked. Formally, the environment can be defined as a finite automaton whose actions are labelled with transactions.

4.3.2 The Database and the Transactions

Initialising the database. In the initial state of the model, the database is empty. The model generates all the possible database states allowed by the application logic, as the environment nondeterministically initiates transactions.



(a) the transactions (b) interleaved executions (c) resource token abstraction

Figure 3: The effect of a resource token on scheduling two transactions.

(a) a true model (b) utilising deadlocks

Figure 4: Modelling a rolled back transaction. Solid arrows denote committed transactions that lead from one persistent state to another. A rollback (dotted arrow) leads back to the originating persistent state, or to an artificial deadlock.

Symmetry reduction of transaction parameter domains and object identifiers. Identifier values model objects references in the application code and surrogate keys in the database, such as item numbers. Symmetry reduction [14] can lead to exponential savings by exploiting the fact that the actual values of these identifiers are irrelevant.

Static analysis can determine the set of operations performed on each domain. Symmetry reduction is only compatible with assignment and equality test. For instance, integer arithmetics requires a (limited) domain of integers or equivalence classes.

For each identifier domain, the model contains a pool of available values.

Transactions and resource tokens. Since the database management system isolates transactions from each other, the transactions can be modelled to be mutually exclusive. This can be arranged by introducing a *resource token* [11] that must be "possessed" by the active transaction.

Figure 3 illustrates the effect of a resource token. There are two enabled transactions, consisting of 2 and 3 operations. Depending on the order in which the operations in the transactions are performed, the system will follow different paths to the final state, shown rightmost in Figures 3(b) and 3(c). Only the corner states of the depicted lattices are *persistent*, meaning that the database is in a committed state. Some of the *transient* states have been eliminated in Figure 3(c).

The resource token abstraction may interfere with partial order reductions [9]. Those techniques work best when the processes in the system are as independent as possible. The resource token makes all transactions depend on each other. Also, verifying liveness properties requires a strong fairness assumption for the first transition of each transaction and a weak fairness assumption for the transitions that return the resource token. The model checker algorithm in MARIA [15] manages these assumptions in an efficient way.

Rolling back transactions. When a transaction is rolled back, the requested changes to the persistent data must be ignored. This can be accomplished in two ways (Figure 4):

(a) by restoring the persistent data from a back-up copy, or

(b) by setting a "rollback" flag that disables all transitions in the model—an artificial deadlock.

Translating rolled back transitions to deadlocks simplifies both the model and its state space. In a real system, rolling back a transaction should restore the database to its original state, as depicted in Figure 4(a). In exhaustive state space enumeration, all reachable states of the system are considered, and deadlock states pose no problem. The search algorithm can still distinguish genuine deadlock states of the system from these artificial deadlocks by examining the "rollback" flag.

4.3.3 Components and Application Code

Mapping objects to relations. There are two types of data in enterprise applications. The transient data that is being processed is managed in objects, while the persistent data in the database is stored as tuples from relational calculus. The models of the components that provide mappings between tuples and objects must address the following issues:

object identifiers: Compared to the relational data model, the object model adds a level of indirection in the form of object identifiers. A unique identifier or reference is assigned to each created object. When an object is no longer needed, the identifier can be freed. The dynamic allocation of identifiers can lead to a combinatorial explosion unless some reduction techniques are applied. Our model limits the explosion by purging all objects and identifiers upon entering a persistent states.

existence tests: Databases are often tested for the existence of records. For instance, the component dbOrder introduced in Section 3.2 must determine whether the customer has an open order, and place a new order if necessary. In Petri nets, transitions are enabled if enough items exist in their input places. Defining an action for the case when something is absent requires a modelling trick, such as using a complement place or a counter, or reserving a special value for denoting absent items.

aggregate operations: Sometimes it is necessary to perform an operation on a group of data, such as all items that belong to an invoice. The total invoiced amount is the sum of the prices of the ordered items multiplied by the ordered quantities. When an invoice header is deleted, the invoice lines listing the billed quantities and identifying the items are deleted as well. This kind of operations can be modelled in high-level nets by making use of inhibitor arcs, as Billington demonstrates [3, Chapter 8], or by introducing auxiliary attributes that can be used to limit dynamic quantifications in the MARIA net class [16]. For instance, there could be a derived place that maps invoice identifiers to invoice line counts.

Components and their composition. Component services can be modelled as transitions that define the effect of invoking the service interface. Nondeterminism can be modelled by defining conflicting transitions for a service. We call this kind of model elements *simple components*.

Transitions can be difficult to derive automatically, if the logic of the program code is complicated. This limits the use of simple components. More complicated cases can be maintained manually as discussed in Section 3.4. Another possibility is to create composite component models. They are derived automatically from the application code. Each statement in the application code is assigned a program counter value within the composite component. A statement corresponds to a transition that performs a computation step and updates the program counter.

Composite components allow program logic to be extracted automatically from the application code. The program counter values increase the state space, even though the counter is reset when the transaction is completed. However, this information is relevant when mapping an error trace to application code statements. The source code file names and line numbers can be encoded either in enumerated program counter values or in transition names.

Simple components do not need program counters. Thus, they can be composed with the rest of the application model by transition substitution. Modelling component execution with a single transition does not introduce intermediate states in the same way as using a program counter does.

Path compression and nondeterministic choices. Eliminating interleavings with the resource token, as illustrated in Figure 3, can result in some non-branching state sequences in the state graph. Such sequences can be collapsed by applying path compression [17].

Nondeterministic components and conditions within application code introduce branches in the state space. The branch target states cannot be eliminated by path compression. However, MARIA is able to distinguish "visible" and "hidden" states. Only the visible states, corresponding to the persistent states of the model, need to be permanently stored.

Eliminating input validation code with static analysis. Typically, application code validates its input. Nearly half the code in Figure 2(a) deals with erroneous input. This code can be omitted from the formal model if the environment is constrained in such a way that it sends only such parameter combinations to the method that would pass the validation. This may lead to significant reductions at the cost of additional static analysis.

Method calls. Object-oriented programs typically contain a large number of method calls. When a *virtual method* is called, the run-time system must determine the type of the object and *dispatch* the call to the applicable method. Sometimes the call target can be determined at compilation time.

The translation of virtual method calls can be simplified by generating a dispatcher method for each virtual method. The dispatcher contains a `switch` block that branches according to the type of the object. In each branch, the dispatcher jumps to a method of a derived class. In this way, each virtual method invocation can be implemented as a non-virtual call to a dispatcher procedure.

Method calls involve some overhead of storing return addresses and copying parameters. For short methods, it is more efficient to substitute calls to the method with the program code in the method body. This technique is referred to as *inlining*. It can eliminate trivial intermediate states, but it may also produce significantly bigger models. In essence, it is a tradeoff between the model size and the number of reachable states.

Folding. Some entities can be modelled as a single high-level Petri net place or as a collection of simpler places. The choice whether to fold may affect the space and time requirements of state space enumeration. Folding places adds flexibility to transitions.

For instance, when the control flow of a program is modelled with a single high-level "program counter" place, a `switch` statement can be translated into a single transition that jumps to one of the `case` labels. If there was a separate program counter place for each statement in the program, the program flow might be more clearly visible from a graphical presentation of the net, but translating the `switch` statement would require more transitions, in fact one for each `case` label.

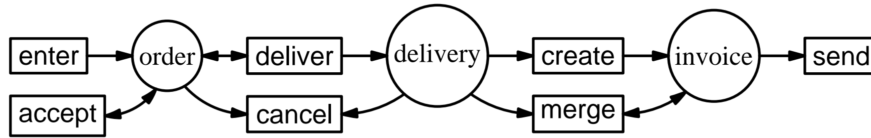


Figure 5: An abstract view of an order processing application.

Similar choices can be made in data type definitions. When a class hierarchy is translated to a single data type definition, objects of a base class can be stored in the same place, no matter which derived class it belongs to. Defining separate data types for derived classes requires a set of places (and transitions) for each derived class.

5 Analysing the Example Application

To evaluate the feasibility of the presented approach, we manually constructed a high-level Petri net model for our example application that was introduced in Section 3.2.

Figure 5 presents a simplified view of the main information flows of the application as a Petri net like graph. The processing starts when an order is entered into the system. Deliveries are controlled by a separate system, to which the order processing system sends a delivery request message, once the order has been accepted.

The delivery system informs the order processing system of completed deliveries. Either system may also initiate a procedure to discard the order and the delivery request.

A delivery confirmation message is transformed into an invoice that will be sent later. If there is an unsent invoice for the customer who made the order, the delivery is merged with this invoice. The last step in the processing chain is to send the invoice to the customer.

5.1 The Model of the Demo Application

In the generated model of the application, each transaction comprises a simple component. Since there are no program counters, all reachable states of this model are persistent database states.

This model was hand crafted, and some abstractions were made. Most notably, the database tables "customer" and "item" were eliminated, because they do not control the behaviour of the transactions we are interested in.

The implementation of the application contains functions for entering and updating information that does not control the application logic, such as names, addresses and prices. Without loss of generality, the domains of these data fields were restricted to one value, which essentially removes the fields from the formal model.

The "order" table contains, among others, three columns for quantities: the quantity of ordered items, the quantity of delivered items, and the quantity of items that have been invoiced. The last column is redundant, as its data can be derived from deliveries and invoices. Databases sometimes contain redundant information, either because deriving the information is computationally too expensive or because the data used for deriving the information might be cleaned up later from the live database to a data warehouse system. Such redundancy could be detected in static analysis,

which may be expensive. On the other hand, eliminating redundant fields does not reduce the number of reachable states, but the space needed for representing a state.

Invoices are stored in two tables. The “invoice row” table links deliveries to the header table “invoice.” In the implementation, the invoice rows are numbered, so that invoices can be retrieved in a consistent order. While the order of invoiced items may be relevant in printed documents, it does not matter in our formal analysis. Therefore, the row number column was abstracted away.

The resulting model in MARIA format [18] has 12 transitions and 10 places. Four places correspond to the modelled database tables. The markings of the remaining six places are functions of the database contents. Three places are identifier pools of unassigned order, delivery and invoice numbers and one place counts the lines belonging to each invoice. Two places—which would be connected to the transitions create and merge depicted in Figure 5—indicate which customers have unsent invoices and which do not.

5.2 A Usage Scenario

In this example scenario, an application coder wants to verify that all the referential integrity rules are respected, and that an order entered will eventually be processed. Processing an order means that the order is delivered and invoiced, or it is cancelled.

The referential integrity rules are translated into safety properties, and the liveness requirements are specified in LTL. Both are checked on the fly by the MARIA tool.

The application designer is likely to begin the analysis of the model by assigning all data domains the cardinality 1. In this configuration, some transactions are permanently disabled. For instance, the transition merge of Figure 5 cannot be enabled unless there may be multiple orders and deliveries. MARIA can detect and report dead transactions.

Next, the user might want to enlarge some domains in order to enable more behaviour in the model. Increasing the cardinalities may reveal spurious errors. For instance, when the database accepts multiple orders but only has room for one invoice, it will be impossible to invoice all deliveries unless they can be combined to the single invoice.

Verifying high-level liveness properties is an interactive procedure where the domain sizes, fairness assumptions and the environment need to be adjusted if an unjustified error is reported.

5.3 Some Results

As Table 1 shows, the state space of the model grows significantly when any of the domains is enlarged. Some of the growth is inherent in the application, as discussed in Section 4.3.1, but much of it is due to the lack of symmetry reduction in the tool we used. Because the system behaviour does not depend on actual data values, exploiting symmetries could lead to exponential savings.

Some domains have a greater impact on the state space size than others. If the system accepts at most one order, it does not matter much how many customers there are who can place the order or how many items are available to be ordered. But as soon as there can be multiple orders and deliveries, the state space explosion breaks loose.

In Table 1, not all parameters of the system are varied. Orders are never cancelled, and the database has room for only one invoice. The system has a large state space, and only parts of it can be viewed at a time. When one parameter is incremented, other parameters must be limited and some transactions may need to be disabled. Obviously, not all errors can be guaranteed to be

Table 1: Sizes of reachability graphs generated by the model without and with path compression reduction when at most one invoice can be generated and orders cannot be cancelled. Increasing the cardinalities of orders and deliveries (O), customers (C) or items (I) affects the numbers of reachable states $|V|$ and transition occurrences $|E|$.

O=1		Original		Reduced	
C	I	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
1	1	16	19	7	11
1	2	31	46	13	29
1	3	46	81	19	55
1	4	61	124	25	89
2	1	43	58	25	41
2	2	85	148	49	113
2	3	127	270	73	217
2	4	169	424	97	353
3	1	82	117	46	82
3	2	163	306	91	235
3	3	244	567	136	460
3	4	325	900	181	757
4	1	133	196	73	137
4	2	265	520	145	401
4	3	397	972	217	793
4	4	529	1,552	289	1,313

O=2		Original		Reduced	
C	I	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
1	1	427	986	409	1,003
1	2	1,609	4,616	1,537	4,591
1	3	3,547	12,042	3,385	11,915
2	1	2,665	7,376	2,521	7,279
2	2	10,369	38,432	9,793	37,759
2	3	23,113	106,992	21,817	105,263
3	1	8,227	26,118	7,741	25,595
3	2	32,329	145,368	30,385	142,703
3	3	72,307	419,958	67,933	413,531

O=3		Original		Reduced	
C	I	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
1	1	14,680	49,341	14,518	50,809
1	2	107,983	447,870	106,687	451,345
2	1	194,923	794,226	192,331	798,889
2	2	1,496,197	8,197,284	1,475,461	8,175,073

found in this kind of analysis, but even partial verification has better coverage than testing. None of the data integrity rules built in the model are violated in the combinations we checked.

6 Related Work

Modelling database systems with Petri nets is nothing new. One earlier method is NetCASE [19], a Petri net based computer aided software engineering (CASE) technique that covers everything from requirements analysis to code generation. It may be hard to apply this kind of methods in practice, where things tend to be built on top of existing systems. We believe in automated reverse engineering, the opposite of code generation.

The PathStar project at Bell Labs [10] showed that a programming language can be treated as a formal model, provided that the source code is annotated appropriately for an automated translator that makes suitable abstractions. In that project, verification experts translated requirement specifications from English prose to LTL and maintained the abstraction rules of the translator, so that it was possible to model check the software under development on a daily basis.

The Bandera [5] and SLAM [2] toolkits create abstract verification models from source code. Bandera inputs the abstractions from the user, while SLAM iteratively refines them by itself. Neither tool seems to support the composition of derived models with hand-crafted fragments.

Lie et al. [13] present a method for automatically extracting models from low level software implementations. The extracted model is combined with a model of the hardware. Their approach

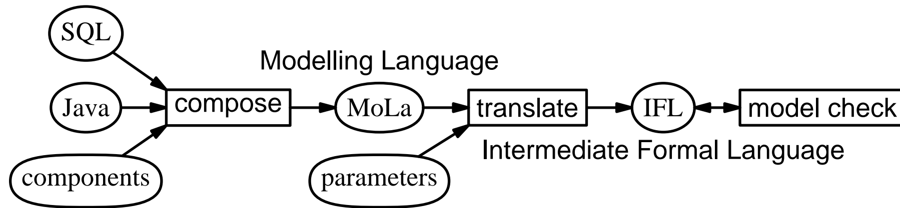


Figure 6: A block diagram of the proposed tool. The prototype will be based on MARIA, but our Intermediate Formal Language can be easily interfaced with other model checkers.

is similar to ours, except that we combine models extracted from high level program code with abstract models of software components.

7 Conclusion and Future Work

Component based software systems are expected to create a flexible and economical infrastructure where companies have a considerable choice of procurement to create customised solutions. When components can be deployed and updated individually, system acquisition and modifications should become more manageable than before. With a simple example, we demonstrated how these data-centric applications are constructed and what their environment looks like.

The architectural style of component-oriented applications, where functionality is hidden behind high-level interfaces, creates an opportunity for applying formal methods, such as state space analysis. Our approach is based on extracting a formal system model from the models of software components and from the application code which glues the components together. This model is formally checked for desired or undesired properties.

Adopting advanced software engineering techniques, such as model checking, in an industrial setting requires well integrated and automated tool support. We propose a tool that allows software maintainers to verify the correctness of systems before system level testing. The objective of this verification step is to gain more insight than could be achieved by pure static analysis techniques.

This tool, depicted in Figure 6, transforms application code, database schema and a repository of component models into a verifiable model of the system. Many desired properties of the system are derived automatically from database definitions and assertions in the application code. Some safety guards, such as the identifier pool alert, are optional. Verifying high-level liveness properties is likely to be an interactive procedure, where the user is required to control the fairness assumptions and the model parameters, such as input domain sizes, if an unjustified error trace is reported. If errors are found, they are presented in terms of the application code.

The application behaviour is mapped to a formal model based on shared memory multiprocessing. In the model, the shared memory is the database and the competing processes are the transactions initiated by the environment. The structure of the application is restricted in such a way that the relations between transaction parameters and database contents can be derived automatically. Each group of related fields and parameters is assigned an own domain.

The state explosion problem is tackled from two directions. Primarily, we rely on the user managing the parameters of the model. Secondly, we build the model in such a way that state space reduction can be accomplished in verification tools.

The state explosion problem can be alleviated by keeping the data domains small. Minimising the data domains could result in some of the application behaviour missing from the model. Here we rely on the user insight and allow him to individually select the sizes of various data domains. The developer may also specify how the environment should behave: which transactions should be invoked and in which order. In this way, users can generate state spaces revealing different aspects of the application behaviour. This partial verification resembles testing, but it can have better coverage.

Our modelling framework abstracts from the inner workings of database management systems. Only one database transaction is processed at a time. Ideally, we would like to store only the persistent database states and the transitions between these states. The state explosion can also be attacked with symmetry reduction [14]. It relies on the fact that the actual values of identifiers are irrelevant, as long as only assignments and equality tests are applied to them. These conditions can be checked by the tool that constructs the verifiable model.

We believe that the proposed tool could help in reducing application maintenance costs. Savings are possible if some of the otherwise required testing can be substituted with verification runs. Applying formal methods to component systems gives a profoundly different starting point for third-party component markets. A formal model—an abstract description of a system—can be thoroughly analysed by computer tools to increase confidence in the system working according to the specification. Without such confidence, customers are easily locked in ordering all further development from the original system vendors.

This article describes "work in progress." Sections 4 and 5 were mainly written by the second author, while the idea of applying state space analysis to component-based software originated from the first author who is preparing his licentiate's thesis on the subject. His plans include writing a front-end for the MARIA tool [16] and using it in simulated application maintenance work. If the results are positive, it will be most interesting to find industrial applications and to see how our approach could be augmented by modelling the business processes [1] and database performance [4]. Also, conformance testing of components [8] could be implemented in the framework.

Acknowledgements

The authors would like to thank Charles Lakos—especially regarding the identifier pool alert—and Nisse Husberg and the anonymous referees for their comments and suggestions.

References

[1] Wil M. P. van der Aalst. Making work flow: On the application of Petri nets to business process management. In Esparza and Lakos [7], pages 1–22.

[2] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In Matthew B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, Toronto, Canada, May 2001. Springer-Verlag.

[3] Jonathan Billington. Extensions to coloured Petri nets and their application to protocols. Technical Report 222, University of Cambridge, Computer Laboratory, Cambridge, England, May 1991.

[4] Ing-Ray Chen and Rajakumar Betapudi. A Petri net model for the performance analysis of transaction database systems with continuous deadlock detection. In Hal Berghel, Terry Hlengl and Joseph Urban, editors, *Proceedings of the 1994 ACM symposium on Applied computing*, pages 539–544, Phoenix, AZ, USA, March 1994. ACM Press, New York, NY, USA.

[5] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander Wolf, editors, *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. ACM Press, New York, NY, USA.

[6] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In Barry Boehm, David Garlan, and Jeff Kramer, editors, *Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, Los Angeles, CA, USA, May 1999. IEEE Computer Society Press, Los Alamitos, CA, USA.

[7] Javier Esparza and Charles Lakos, editors, *Application and Theory of Petri Nets 2002, 23rd International Conference, ICATPN 2002*, volume 2360 of *Lecture Notes in Computer Science*, Adelaide, Australia, June 2002. Springer-Verlag.

[8] Leonard Gallagher. Conformance testing of object-oriented components specified by state/transition classes. Draft technical report, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, USA, April 6, 1999.

[9] Patrice Godefroid, Doron Peled and Mark Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on Software Engineering*, 22(7):496–507, July 1996.

[10] Gerard J. Holzmann and Margaret H. Smith. Software model checking: extracting verification models from source code. *Software Testing, Verification & Reliability*, 11:65–79, 2001.

[11] Nisse Husberg and Tapio Manner. Emma: developing an industrial reachability analyser for SDL. In Jeannette M. Wing, Jim Woodcock and Jim Davies, editors, *World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pages 642–661, Toulouse, France, September 1999. Springer-Verlag.

[12] Kuldeep Kumar and Jos van Hillegersberg. ERP experiences and evolution. *Communications of the ACM*, 43(4):23–26, 2000.

[13] David Lie, Andy Chou, Dawson Engler and David L. Dill. A simple method for extracting models from protocol code. *Proceedings of the 28th Annual International Symposium on Computer Architecture, ISCA 2001*, pages 192–203, Göteborg, Sweden, July 2001. IEEE Computer Society.

[14] Tommi Junttila. Finding symmetries of algebraic system nets. *Fundamenta Informaticae*, 37(3):269–289, February 1999.

[15] Timo Latvala. Model checking LTL properties of high-level Petri nets with fairness constraints. In José-Manuel Colom and Maciej Koutny, editors, *Application and Theory of Petri Nets 2001, 22nd International Conference*, volume 2075 of *Lecture Notes in Computer Science*, pages 242–262, Newcastle upon Tyne, England, June 2001. Springer-Verlag.

[16] Marko Mäkelä. Maria: Modular reachability analyser for algebraic system nets. In Esparza and Lakos [7], pages 427–436.

[17] Marko Mäkelä. Efficiently verifying safety properties with idle office computers. In Charles Lakos, Robert Esser, Lars M. Kristensen and Jonathan Billington, editors, *Formal Methods in Software Engineering and Defence Systems 2002*, volume 12 of *Conferences in Research and Practice in Information Technology*, pages 11–16, Adelaide, Australia, June 2002. Australian Computer Society Inc.

[18] Marko Mäkelä. Maria model of the Jive demo application. `http://www.tcs.hut.fi/maria/samples/jive/`.

[19] Thomas Marx. NetCASE—a Petri net based method for database application design and generation. Research report 11-95, University of Koblenz, Germany, September 1995.

[20] Roger S. Pressman. *Software Engineering: A Practitioner's Approach, European Adaptation*. McGraw-Hill International Editions, 2000.

[21] Bill Shannon. *Java 2 Platform, Enterprise Edition: Platform and Component Specifications*. Addison-Wesley, 2000.

[22] David Sprott. Componentizing the enterprise application packages. *Communications of the ACM*, 43(4):63–69, 2000.

[23] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems. Volume I: Classical Database Systems*. Computer Science Press, 1988.

Modelling with Coloured Petri Nets

Søren Christensen
Department of Computer Science
University of Aarhus
Denmark

Abstract

Engineers have used construction of models to investigate properties of designs, before implementations, for generations. This trend is gradually being transferred to the software engineering. Today UML is well established in the modelling of applications where the structure of data is the main concern, e.g. traditional office applications and applications for database access. During the process of creating and testing the models the designers gain insight which allow them to improve their designs and try different solutions before crucial design decisions are made.

Coloured Petri Nets and the tools Design/CPN and CPN Tools offers a supplement to the mainly data driven models of UML, and we will show examples of models where Coloured Petri Nets are used to capture essential behavioural properties of systems. We address important issues of the modelling process, such as: finding the right level of abstraction, how to structure large models and what not to model.

For the modelling to be successful it is equally important that we have rich formalisms, powerful computer tools, and skilled engineers. Only this combination will allow us to build the increasingly complex systems needed in the future.

Executable Use Cases for Pervasive Healthcare

Jens Bæk Jørgensen and Claus Bossen
Centre for Pervasive Computing
Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
email: jbj@daimi.au.dk, bossen@daimi.au.dk

Abstract

Using a pervasive healthcare system as example, a new approach to specification of user requirements for pervasive IT systems is presented. A formal modelling language, Coloured Petri Nets, is applied to describe what we call Executable Use Cases, EUCs. EUCs are precise, detailed, and executable descriptions of future work processes and their computer support. In particular, EUCs allow user requirements specifications to take the frequently changing context of the users, e.g. their location and equipment in possession, into account.

Topics: Specification of user requirements, use cases, Unified Modeling Language (UML), application of Coloured Petri Nets to pervasive systems, context awareness.

1 Introduction

Worldwide, many hospitals are in the process of introducing *electronic patient records, EPRs*, or have already done so [1]. Though a vision of an EPR already appeared in the late 1960s, IT technology apparently only reached a level that enabled workable solutions in the 1990s, when the spread of EPRs began to gain momentum [6]. In Denmark, Aarhus County has initiated development of an EPR [18] that will substitute several paper-based core documents used for documentation and communication within hospitals today. The aim is to enhance the quality of healthcare, e.g. by allowing multiple users to work on the same, always up-to-date patient record at the same time.

The EPR of Aarhus County, and indeed any EPR, solves obvious problems occurring with paper-based patient records such as being not always up-to-date, mislaid, or even lost. However, EPRs also have their drawbacks and potentially induce at least two central problems for their users. The first problem is *immobility*: in contrast to a paper-based record, an EPR accessed from stationary desktop PCs cannot be easily transported. The second problem is *time-consuming login and navigation*: EPR requires user identification and login to ensure information confidentiality and integrity, and to start using EPR for clinical work, a logged-in user must navigate, e.g. to find a specific document for a specific patient. *Pervasive computing* [2, 7, 8] is a candidate approach to alleviate these two problems. Specifically, a new *pervasive healthcare system* [3, 21]

is being envisioned in a joint project between Aarhus County Hospital, the software company Systematic Software Engineering A/S [23], and the Centre for Pervasive Computing [19] at the University of Aarhus. This paper focuses on specification of the *user requirements* for that system.

One of the dominant approaches to specify user requirements in today's object-oriented system development projects is *use cases* [4, 9, 12] as defined in the Unified Modeling Language, UML [13, 15]. Use cases model work processes to be supported by a new IT system, and a set of use cases is interpreted as user requirements for that system. However, UML use cases have several general and known shortcomings, see e.g. [16] which points out a number of problems under headlines like *use case modelling misses long-range logical dependency* and *use case dependency is non-logical and inconsistent.* For pervasive systems, new complexities are added to the specification of user requirements in order to cope with issues like mobility and context awareness, i.e. the ability of the IT system to react sensibly to various changes of context such as users moving from one location to another.

For these reasons, we will suggest a new notion of use cases. Inspired by the UML use case approach, we will also create models of work processes, but will do so in the formal modelling language *Coloured Petri Nets, CPN* [10, 11, 20]. CPN models are precise, detailed, and, as a particularly valuable asset, executable. Furthermore, as we will show, CPN is suitable to describe *context aware systems*. Contexts can be captured by an elaborated state notion of CPN, and the functionality of a system in a current context can be modelled by an appropriate action notion of CPN. The main contribution of the paper is to introduce and justify the notion of *Executable Use Cases, EUCs*, based on CPN, for the specification of user requirements for pervasive IT systems.

The paper provides a basis introduction to CPN, and, thus, does not assume the reader to be familiar with neither CPN, nor Petri nets [14] in general. The structure is as follows: Section 2 describes the concept of *pervasive healthcare* and the *pervasive medicine administration* work process that will be used as running example. Section 3 is a primer on CPN, and Section 4 presents an EUC for pervasive medicine administration. In Section 5, we report on how to interpret an EUC as actual user requirements. The conclusions are drawn in Section 6.

2 Pervasive Healthcare

The pervasive healthcare system considered in this paper was envisioned in a series of workshops with participation of physicians, nurses, computer scientists, and an anthropologist (one of the authors). Moreover, input came from ethnographic fieldwork by the anthropologist, who spent two months at a department at Aarhus County Hospital in spring 2001. Observation of existing work processes and use of the paper-based patient records confirmed that immobility and time-consuming login and navigation procedures are severe, potential obstacles to the success of EPR.

2.1 Characteristics of Hospital Work Processes

The EPR immobility problem should be solved in order to preserve the inherent mobility of hospital work processes. Paper patient records are very mobile and now frequently moved, e.g. between the nurses' office, the medicine cabinet room, and the wards with patient beds. The other anticipated problem, time-consuming login and navigation, is crucial to overcome in order to allow healthcare personnel to make flexible and smooth transition from one work process to another. This is needed, because interruption and later resumption of work processes are frequent: the personnel continuously have to reschedule their plans for their shift, since new tasks are often added to their list of what to do. New, acutely ill patients are admitted to the department, examinations show a need for immediate action, the condition of a patient deteriorates suddenly, scheduled examinations at other departments are cancelled, etc.

The present plans for EPR deployment entails an additional, severe problem: restricted access to the patient records. The paper-based patient record comprises three separate documents for each patient, a physician's patient record, a nurse's patient record, and a medicine plan. These documents are kept at different places most of the day. Thus, even though only one person can access a specific part of a patient record at a time, there are in the considered department, with 25 patients, nevertheless 75 different access points to the records (three per patient). With EPR, accessibility is proportional with the number of available computers – an obvious bottleneck. The department at present budgets for eight desktop PCs to be placed in offices and two laptops which can be brought along to the wards. In this way, the existing 75 access points decrease to only 8+2=10.

The work process in focus in this paper is *medicine administration*, i.e. handling of medicine for patients. It involves *medicine plans*, which for each patient specify the prescribed medicine, and are used by the nurses to acknowledge when medicine has been poured and given. Medicine plans are usually kept by the medicine cabinet and often taken to the wards together with the medicine. This allows nurses to promptly acknowledge giving of medicine at the wards and to answer questions from patients about their medication. To enable a smooth medicine administration process with EPR, possible solutions with today's technology seem to be computers in each ward or widespread use of personal digital assistants, PDAs – laptops are too heavy to be regularly carried around.

2.2 Pervasive Healthcare System Design Principles

A prototype of the pervasive healthcare system has been created and subsequently tested by healthcare personnel from Aarhus County Hospital [3]. The prototype is built upon three general design principles. The first principle is *context awareness*. This means that the system is able to register and react upon certain changes of context. More specifically, nurses, medicine trays, patients, beds, and other items are equipped with radio frequency identity, RFID, tags [22], such that presence of such items can be detected automatically by involved context aware computers, e.g. located by the medicine cabinet and by the patient beds.

The second design principle is that the system is *propositional*, in the sense that it makes qualified propositions, or guesses. Context changes may result

in automatic generation of buttons, which appear at the taskbar of computers. Users must explicitly accept a proposition by clicking a button – and implicitly ignore or reject it by not clicking. The presence of a nurse holding a medicine tray for patient P in front of the medicine cabinet is a context that triggers automatic generation of a button `Medicine plan: Patient P`, because in many cases, the intention of the nurse is now to navigate to the medicine plan for patient P. If the nurse clicks the button, she is logged in and taken to patient P's medicine plan. It is of course impossible always to guess the intention of a user from a given context, and without the propositional principle, automatic shortcutting could become a nuisance, because of guesses that would sometimes be wrong.

The third design principle is that the system is *non-intrusive*, i.e not interfering with or interrupting hospital work processes in an undesired way. Thus, when a nurse approaches a computer, it should react on her presence in such a way that a second nurse, who may currently be working on the computer, is not disturbed or interrupted. The last two design principles cooperate to ensure satisfaction of a basic mandatory user requirement: important hospital work processes have to be executed as conscious and active acts by responsible human personnel, not automatically by a computer.

2.3 Pervasive Medicine Administration

Work process descriptions in natural language were made as a result of the workshops, and these descriptions formed the basis for the prototype discussed above. The work process constituting the scope of this paper is *pervasive medicine administration*. It covers medicine administration as carried out by personnel supported by the pervasive healthcare system, and is outlined in the following in a style resembling a main constituent of a traditional UML use case.

Assume that nurse N wants to pour medicine into a medicine tray and give it to patient P. First, the nurse goes to the room containing the medicine cabinet. Here is a context aware computer on which the buttons `Login: Nurse N` and `Patient list: Nurse N` appear on the taskbar, when the nurse approaches. Assume that the second button is clicked. Then, N is logged in and a list of those patients for which she is in charge is displayed on the computer.

A medicine tray must be associated with each patient. If a medicine tray is already associated with patient P, the button `Medicine plan: Patient P` will appear on the taskbar of the computer, when the nurse takes the tray nearby, and a click will make P's medicine plan appear on the display. However, if P is a newly admitted patient, it is necessary to associate a medicine tray to him. Nurse N does so by taking an empty tray from a shelf and making the association. In either case, N pours medicine into the tray, acknowledges this in EPR, and is automatically logged out, when she leaves the medicine cabinet area.

Nurse N now takes patient P's medicine tray and goes to the ward where P lies in a bed, which is supplied with a context aware computer. When N approaches, the buttons `Login: Nurse N`, `Patient list: Nurse N`, and `Medicine plan: Patient P` will appear on the taskbar. If the last button is clicked, the medicine plan for P is displayed. Finally, N gives the medicine tray to P, acknowledges the giving in EPR, and is automatically logged out again, when she leaves the bed area.

This rather straight flow of events has numerous variations, e.g. medicine may be poured for one or more patients, for only one round of medicine giving, all four regular rounds of a 24 hours period, or for ad hoc giving; a nurse may have to fetch trays left at the wards prior to pouring; a nurse may approach the medicine cabinet without intending to pour medicine, but only to log into EPR or wanting to check an already filled medicine tray. To enable a smooth pervasive medicine administration work process, the pervasive healthcare system must be specified to handle all these variations and many more. The Executable Use Case, EUC, apparatus is able to do that. We will present an EUC for pervasive medicine administration in Section 4, but first give an informal and general introduction to some fundamental concepts of the EUC modelling language CPN in the following section.

3 EUC Modelling Language – CPN

Coloured Petri Nets, CPN [10, 11, 20], is a mature and well-proven modelling language suitable to describe the behaviour of systems with concurrency, resource sharing, and synchronisation. A CPN model resembles a board game, with strict rules that define the possible executions of the model. The CPN modeller's task is to specify an appropriate board, tokens, and playing rules to reflect the domain being modelled.

We will introduce the basic CPN concepts by means of the simple `Provide trays` model, shown in Figure 1. This model describes how nurses provide medicine trays prior to pouring, checking, and giving medicine.

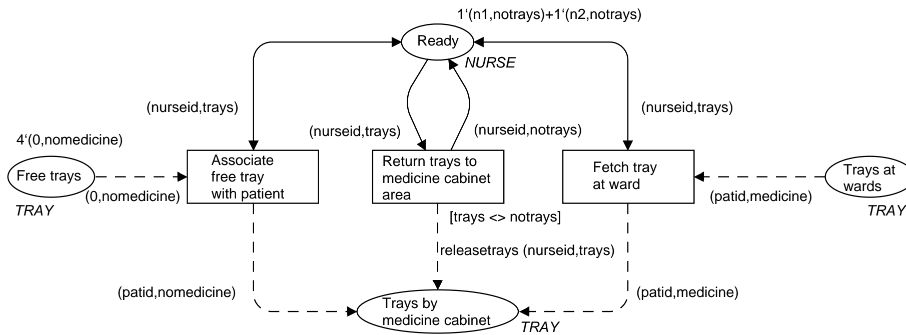


Figure 1: `Provide trays`.

3.1 Modelling of States

A CPN model describes both the states and the actions of a system. States capture the contexts in which actions may take place. The *state* of a CPN model is a distribution of *tokens* on the *places*. Each place is drawn as an ellipse and has an associated *data type*, written in italic capital letters, which determines the kinds ("colours") of tokens the place is allowed to contain.

In Figure 1, a token on the `Ready` place models that a real-life nurse is in a situation where she is ready to carry out work, e.g. she sits at her office.

Thus, `Ready` has the data type `NURSE`, denoting nurses. A nurse is represented as a pair `(nurseid,trays)`, where `nurseid` identifies the nurse and `trays` is a container data structure holding the medicine trays that this nurse currently has in possession.

The places `Trays at wards` and `Trays by medicine cabinet` have data type `TRAY`, and model the trays which currently are at the indicated locations. A tray is modelled as a pair `(patid,medicine)`, where `patid` identifies the patient that the tray is associated with, and `medicine` is a container data structure holding the medicine that currently is in the tray. The place `Free trays` also has data type `TRAY`. It holds tokens corresponding to trays which are not yet associated with patients. The patient id 0 is a special value, used in `TRAY` tokens on the form `(0,nomedicine)`, which represent empty trays, not currently associated with any patient.

The *initial state* describes the start state of the model, before execution begins. In the initial state, there are two `NURSE` tokens on the `Ready` place, `n1` and `n2`, both with no trays, and four empty, non-associated `TRAY` tokens on the `Free trays` place, as indicated by the inscriptions close to the places (an expression like `c1‘e1` denotes a multiset (a bag) containing `c1` appearances of `e1` tokens). The places `Trays at wards` and `Trays by the medicine cabinet` are empty in the initial state.

3.2 Modelling of Actions

The *actions* of a CPN model are represented using *transitions*, drawn as rectangles. Thus, in Figure 1, a nurse who is ready – corresponding to a token in the `Ready` place – may choose to do one of three possible actions, modelled by the three transitions named `Associate free tray with patient`, `Return trays to medicine cabinet area`, and `Fetch tray at ward`.

A transition and a place may be connected by an *arc*. Solid arcs show the flow of `NURSE` tokens, and dashed arcs the flow of `TRAY` tokens (different graphical appearances are used only to enhance readability, and have no formal meaning). The actions of a CPN model consist of transitions removing tokens from input places and adding tokens to output places, often referred to as the *token game*. Input/output relationship between a place and a transition is determined by the direction of the connecting arc. A place may be both input and output, e.g. `Ready` relative to all three transitions – a double arc is a shorthand for one arc in each direction. The tokens removed and added are determined by *arc expressions*, e.g. the expression `(nurseid,trays)` on the arc from the `Ready` place to the `Return trays to medicine cabinet area` transition, where `nurseid` and `trays` are variables that can be assigned data values.

The executability of CPN models comes from the fact that the CPN modelling language has a *formal, operational semantics.* A transition which is ready to remove and add tokens is said to be *enabled*, and requires two kinds of conditions to be fulfilled. The first kind is that appropriate tokens are present on the input places. This means that one condition for enabling of the transition `Return trays to medicine cabinet area` is that the only input place, `Ready`, contains some token matching the expression `(nurseid,trays)`. The second kind of condition comes from the *guard*, which is a boolean expression optionally assigned to a transition, and which must evaluate to true for the

transition to be enabled. `Return trays to medicine cabinet area` has the guard `[trays<>notrays]`. This means that a nurse may return trays to the medicine cabinet area, only when she has some in possession.

An enabled transition may *occur*. The occurrence of `Return trays to medicine cabinet area` models that a nurse takes all the trays she is currently possessing and returns them to the medicine cabinet area, and afterwards is ready again. In the CPN model, this is reflected by a token `(nurseid,trays)` removed from `Ready`, a token `(nurseid,notrays)` added to `Ready`, and a token determined by the expression `releasetrays(nurseid,trays)` added to `Trays by medicine cabinet`.

4 Pervasive Medicine Administration EUC

We have now introduced the necessary CPN concepts, allowing us to present an EUC for pervasive medicine administration. The EUC models the work process from the point of view of a nurse, and a prime focus is to describe how the computers which are by the medicine cabinet and by the patient beds react on context changes and how they support the nurses.

In general, a CPN model consists of a number of modules organised in a hierarchical fashion – Section 3 presented just one single module. The pervasive medicine administration EUC CPN model consists of 11 modules, with a total of 54 places and 29 transitions. We will provide a general overview of the model and supplement it with a detailed explanation of one selected typical module.

4.1 Model Overview

An overview of the model in terms of a hierarchy with a node for each module and arcs showing the relationship between the modules is given in Figure 2. The figure shows how the work process pervasive medicine administration is split into sub work processes. An arc between two nodes indicates that the module of the source node contains a *substitution transition*, whose detailed behaviour is described on the module of the destination node, called the *sub-module*.

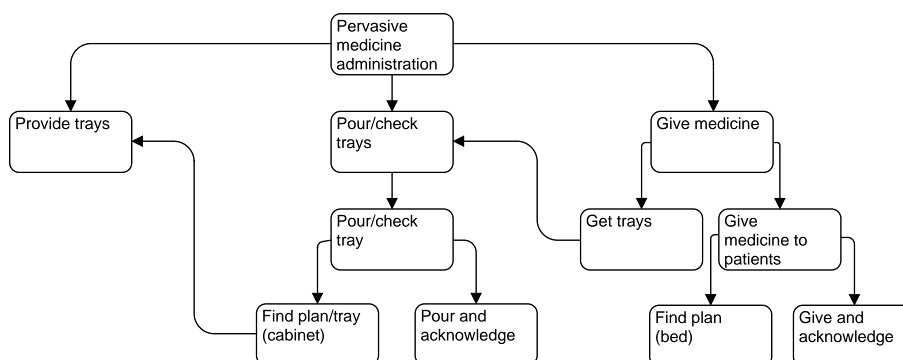


Figure 2: Overview of pervasive medicine administration EUC.

The topmost module `Pervasive medicine administration` of Figure 2 is shown in Figure 3 and contains a very high-level and abstract description of the

work process. All three transitions in Figure 3 are substitution transitions – as indicated by the small `HS` (hierarchy substitution) tag. They correspond to the three arcs emanating from the `Pervasive medicine administration` node in Figure 2, and are briefly described below.

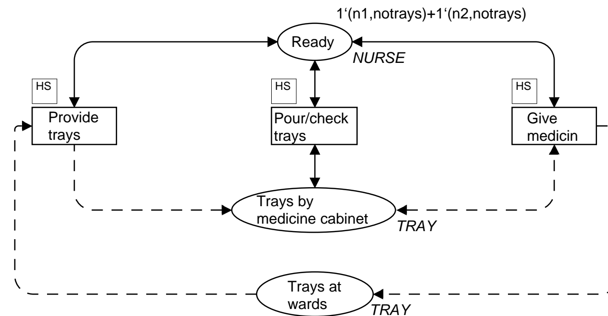


Figure 3: `Pervasive medicine administration` module (topmost node of Figure 2).

4.1.1 Provide trays

We have previously presented and explained this module in Figure 1 in Section 3. From Figure 3, it is possible to see that a nurse sometimes provides trays, by carrying out the action corresponding to the `Provide trays` transition. It is not possible to see how she does it. The sub-module that is bound to the substitution transition `Provide trays`, shown in Figure 1, contains the detailed description of how trays can be provided.

The individual modules of a CPN model interact when the model is executed. In the modules of Figures 3 and 1, places with the same name (e.g. the two `Ready` places) are conceptually glued together, thus allowing exchange of tokens between the modules when the token game is played.

4.1.2 Pour/check trays

As can be seen from Figure 2, the module `Pour/check trays` (note plural 's' in `trays`) uses a sub-module called `Pour/check tray`. `Pour/check trays` models how trays are poured and/or checked for a number of patients. `Pour/check tray` models how a tray is poured and/or checked for one single patient. The module `Pour/check trays` may be seen as similar to a loop statement in a programming language and `Pour/check tray` as the body of the loop.

`Pour/check tray` is itself taking advantage of two sub-modules, one called `Find plan/tray (cabinet)` and one called `Pour and acknowledge`. The first module, `Find plan/tray (cabinet)` models how the nurse gets the medication plan for a given patient presented on the screen of the medicine cabinet computer and/or how she provides the patient's tray – as can be seen on Figure 2, possibly using the `Provide trays` module already described. The `Pour and acknowledge` module models how the nurse actually pours medicine into the tray and how she acknowledges to the pervasive healthcare system when a certain medicine type has been poured.

4.1.3 Give medicine

As can be seen from Figure 2, the module `Give medicine` uses two sub-modules, `Get trays` and `Give medicine to patients`. `Get trays` models how the nurse collects a number of trays (corresponding to a number of patients), before she embarks on a round to the wards to give medicine. This preparation process may involve checking and additional pouring of existing partly or fully filled trays, and therefore `Get trays` uses the `Pour/check trays` module. The `Give medicine to patients` and its two sub-modules `Find plan (bed)` and `Give and acknowledge` are similar to the `Pour/check tray (cabinet)` and its two sub-modules.

4.2 Module Example – `Pour/check trays`

We now present and explain a typical module from the model, the `Pour/check trays` module, shown in Figure 4.

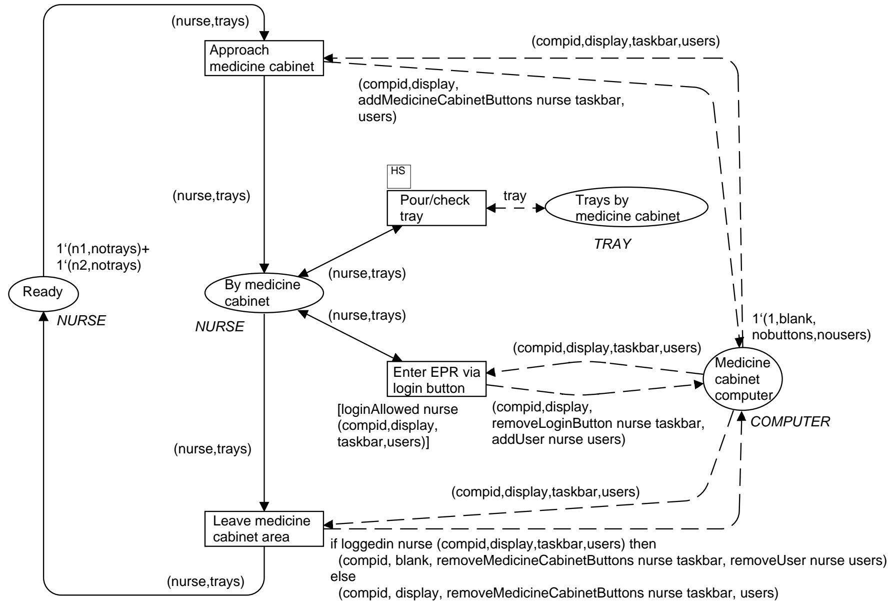


Figure 4: `Pour/check trays` module.

On this module, the medicine cabinet computer is in focus. The computer is modelled by a token on the `Medicine cabinet computer` place having data type `COMPUTER` – a 4-tuple `(compid,display,taskbar,users)` consisting of a computer identification, its display (main screen), its taskbar, and its current users. Long-dashed arcs show the flow of the `COMPUTER` token. Recall that the medicine cabinet computer is context aware, i.e. it is able to sense the proximity of both nurses and trays.

In the initial state, two nurses `n1` and `n2` are ready and have no trays, corresponding to two tokens in the `Ready` place. The medicine cabinet computer is idle, with a blank display, no taskbar buttons, and no current users.

Occurrence of the `Approach medicine cabinet` transition models that a nurse changes from being ready to being busy nearby the medicine cabinet. Moreover, at the same time, two buttons are added to the taskbar of the medicine cabinet computer, namely one login button for the nurse and one patient list button for the nurse. In the CPN model, these taskbar buttons are added by the function `addMedicineCabinetButtons` appearing on the arc going from the transition `Approach medicine cabinet` to the place `Medicine cabinet computer`.

The possible actions for a nurse who is by the medicine cabinet are modelled by the three transitions `Pour/check tray`, `Enter EPR via login button`, and `Leave medicine cabinet area`. Often, a nurse by the medicine cabinet wants to pour and/or check some trays. How this pouring and checking is carried out is modelled on the sub-module `Pour/check tray`, which is bound to the substitution transition having the same name (as can be seen from Figure 2).

The `Enter EPR via login button` transition models that a nurse clicks on the login button and makes a general-purpose login to EPR. It is outside the scope of the model to describe what the nurse subsequently does – the domain of the model is specifically pervasive medicine administration, not general EPR use. The transition has a guard which ensures that only a user who is not currently logged into EPR can do so. When a nurse logs in, the login button for that nurse is removed from the taskbar of the computer, modelled by the `removeLoginButton` function on the arc from `Enter EPR via login button` to the `Medicine cabinet computer` place. Moreover, the nurse is added to the set of current users by the function `addUser` appearing on the same arc.

The `Leave medicine cabinet area` transition models that a nurse has done the pouring and checking that she wants for now, and leaves the area. Upon leaving, it is checked whether the nurse is currently logged in, modelled by the function `loggedIn` appearing in the if-then-else expression on the arc going from `Leave medicine cabinet area` to the `Medicine cabinet computer` place. As can be seen further from the expression on that arc, if the nurse is logged in, the medicine cabinet computer automatically blanks off the screen, removes her taskbar buttons (`removeMedicineCabinetButtons`), and logs her off (`removeUser`). If she is not logged in, the buttons generated because of her presence are removed, but the state of the computer is otherwise left unaltered. When a nurse leaves the medicine cabinet area, the corresponding token is put back on the `Ready` place.

5 EUCs as User Requirements

From the EUC for pervasive medicine administration, a list of user requirements can be produced by focusing on the model transitions which manipulate the involved computers. Each transition connected to the places `Medicine cabinet computer` (shown) and `Bed computers` (not shown) must be taken into account. As examples, the requirements below are induced by the transitions on the module `Pour/check trays` shown in Figure 4:

- When a nurse approaches the medicine cabinet area, the medicine cabinet computer must add a login button and a patient list button for that nurse to the taskbar.

- When a nurse leaves the medicine cabinet area, if she is logged in, the

medicine cabinet computer must blank off its display, remove the nurse's login button and patient list button from the taskbar, and log her off.

- When a nurse logs into EPR, her login button must be removed from the taskbar of the computer, thus disallowing the nurse to log in again, if she already is.

The first two of these three user requirements can also partly be derived from the prose English description of pervasive medicine administration in Section 2.3, i.e. these two requirements were known after the workshops, prior to creation of the EUC. However, the EUC states the requirements more precisely and with more details, e.g. the EUC specifies that the computer display should be blanked when a logged in nurse leaves the medicine cabinet area, and it also describes what should happen when a nurse, who is not logged in, leaves.

In general, a user requirements specification should be precise and address as many issues as possible. We will argue that to a higher extent than reading a static prose descriptions of future work processes as the one in Section 2.3, application of EUCs forces many questions to be asked. An EUC is a dynamic, executable model, allowing the behaviour of a future work process to be visualised and investigated. Playing the token game of an EUC typically catalyses the participants' cognition and generates new ideas. In this way, it often happens that questions that the participants had not thought about earlier appear. Examples of questions (Qs) that have appeared during execution of the EUC of Section 4, and corresponding answers (As), are:

- Q: what happens if two nurses both are close to the medicine cabinet computer? A: the computer generates login buttons and patient list buttons for both of them.

- Q: what happens when a nurse carrying a number of medicine trays approaches a bed? A: in addition to a login button and a patient list button for that nurse, only one medicine plan button is generated – a button for the patient associated with that bed.

- Q: is it possible for one nurse to acknowledge pouring of medicine for a given patient while another nurse at the same time acknowledges giving of medicine for that same patient? A: no. That would require a systems architecture allowing a higher degree of concurrency and a more fine-grained concurrency control exercised over the patient records.

Questions like the ones above may imply changes to be made to the EUC, because emergence of a question indicates that the current version of the EUC does not reflect the future work process properly. As a concrete example, in an early version of the pervasive medicine administration EUC, the exit of any nurse from the medicine cabinet area resulted in the computer screen being blanked off. To be compliant with the non-intrusive design principle, the exit of a nurse who is not logged in, should of course not disturb another nurse who might be working at the computer, and the EUC had to be changed accordingly.

Specification of user requirements is a prominent, general problem in the software industry today. In many development projects, the user requirements are initially too vaguely specified and too poorly understood. This is quite

unfortunate, because a user requirements specification for a software system is often an essential part of a legal contract between a customer, e.g. a hospital, and a software company. Questions like the three above may easily be subject to dispute. However, if the parties have agreed that pervasive medicine administration should be supported, and have the overall stipulation that the EUC presented in the previous section is the authoritative description, many disagreements can quickly be settled, because of the formality and unambiguity of the EUC.

EUCs are not a panacea. Their purpose is solely to describe the user requirements of a future pervasive IT system, relative to the work flow to be supported. A number of other user requirements issues regarding IT systems support of pervasive medicine administration cannot be addressed properly by EUCs, e.g.:

- *User interface*, e.g. where are the buttons placed, what do they look like, and how is a medication plan presented on the computer screen?
- *Response times*, e.g. how long should a nurse wait before her buttons appear on the screen?
- *Distance*, e.g. how close should a nurse be to a computer, before it is aware of her presence?

6 Conclusions

The pervasive healthcare system, and many pervasive systems in general, are characterised by classical and well known complications that apply to many distributed systems [5], plus a number of new problems to be tackled, e.g. regarding context awareness and mobility. With more complex systems come increased demands to the modelling languages that we use for their specification and development. In this paper, the notion of Executable Use Cases, EUCs, based CPN, is introduced. CPN is one dialect of Petri nets [14], and various kinds of Petri nets have previously been used for modelling of work flows [17], much in the same way as in the EUC approach. However, application of Petri nets to specification of user requirements for pervasive systems is, to the best of our knowledge, new.

UML is a de facto standard for object-oriented modelling in the software industry, and UML use cases are very popular for specification of the user requirements for many of today's IT systems. We believe that UML use cases are not always sufficient for pervasive systems. A viable alternative is EUCs, which have three main strengths compared to UML use cases in general: *precision*, *detail*, and *executability*, cf. the comparison in Section 5 of the UML use case style prose description of pervasive medicine administration of Section 2.3 and the EUC of Sect. 4. An additional strength of EUCs and CPN is the rich, elaborated, and precise notion of state, which is not commonly found in other modelling languages, and certainly not in UML. With the state notion, the modelling of contexts comes very natural, e.g. the frequently occurring context element location is conveniently captured by means of CPN places, cf. place names like `By medicine cabinet` in the EUC of this paper. This makes EUCs suitable to specify user requirements for pervasive, context aware systems.

As cited in the introduction, one of the main critiques raised against UML use cases is that they promote a highly localised perspective [16], and do not properly capture dependencies between various sub work processes. For the pervasive medicine administration work process, this general problem has materialised as being difficult and cumbersome to describe with prose text in a sufficiently precise fashion the many dependencies between the involved sub work processes. Such dependencies are explicitly and precisely captured in the EUC, cf. the EUC overview in Figure 2.

EUCs, of course, have potential drawbacks. One drawback is that CPN is indeed a formal language (in other situations, an advantage, though), and thus difficult to use as a means of communication between system developers and non-technical end users. In contrast, one of the often cited strengths of UML use cases is that they are non-formal and easy to understand for users. Like UML use cases, for the best result, EUCs should be worked out with participation of the future system users, in an iterative fashion going from a coarse and probably not entirely correct representation of a future work process, to more and more mature versions of the EUC, where precision and detail are added in each iteration. To ease communication with healthcare personnel, we have started a project to make the EUC CPN model of this paper the logical controller of a small movie-like computer animation displaying work processes at a hospital department with nurses, medicine trays, medicine cabinets, wards, beds, computers, etc. In this way, instead of looking directly at the token game of the CPN model, the nurses and physicians will see their future work situation illustrated in a fashion which is much more natural for them, and where the motions of pictures are controlled by, and thus guaranteed to be consistent with, the execution of the CPN model.

Another drawback of EUCs is that CPN is often thought of as being rather complex and time-consuming to learn and to apply. It is true that it takes an effort to learn. However, once learned, it is quite efficient to apply. The EUC of this paper was created by the authors of this paper using a total effort of approximately 60 man hours, 50 hours used by a computer scientist, who is an experienced CPN modeller, and 10 hours used by the anthropologist who has a detailed knowledge of the work processes at the hospitals. The EUC was created in four iterations where the computer scientist was the active modeller and the anthropologist a participant in executions of the various versions of the EUC, and the main source for questions that provided input to the next, more complete, more detailed, and more precise version of the EUC.

Compared to UML use cases, EUCs do potentially require an extra effort during specification of user requirements. However, we think that in many cases, the investment is well justified. Even though many contracts recognise that the user requirements may change during a project (incurring a price adjustment), it is wiser, cheaper, less frustrating, and more efficient for all parties to properly address as many issues as possible already in the initial specification.

References

[1] M. Berg. Accumulating and Cordinating: Occasions for Information Technologies in Medical Work. In *Computer Supported Cooperative Work*, volume 8, 1999.

[2] J. Burkhardt, H. Henn, S. Hepper, K. Rintdorff, and T. Schäck. *Pervasive Computing – Technology and Architecture of Mobile Internet Applications*. Addison-Wesley, 2002.

[3] H.B. Christensen, J. Bardram, and S. Dittmer. Theme One: Administration and Documentation of Medicine. Report and Evaluation. TR-3. Technical report, Centre for Pervasive Computing, Aarhus, Denmark, 2001.

[4] A. Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2000.

[5] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems – Concepts and Design*. Addison-Wesley, 2001.

[6] R.S. Dick, E.B. Steen, and D.E. Detmer. *The Computer-Based Patient Record: An Essential Technology for Health Care*. National Academy Press, 1997.

[7] U. Hansmann, L. Merk, M.S. Nicklous, and T. Stober. *Pervasive Computing Handbook*. Springer Verlag, 2001.

[8] A. Helal, B. Haskell, J.L. Carter, R. Brice, D. Woelk, and M. Rusinkiewicz. *Any Time, Anywhere Computing – Mobile Computing Concepts and Technology*. Kluwer Academic Publishers, 1999.

[9] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[10] 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.

[11] 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.

[12] D. Kulak, E. Guiney, and E. Lavkulich. *Use Cases: Requirements in Context*. Addison-Wesley, 2000.

[13] OMG Unified Modeling Language Specification, Version 1.4. Object Management Group (OMG); UML Revision Taskforce, 2001.

[14] W. Reisig. *Petri Nets, an Introduction*. Springer-Verlag, 1985.

[15] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[16] A.J.H. Simons and I. Graham. 30 Things That Go Wrong in Object Modelling with UML 1.3. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*. Kluwer Academic Publishers, 1999.

[17] W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[18] Aarhus Amt Electronic Patient Record. www.epj.aaa.dk.

[19] Centre for Pervasive Computing. www.pervasive.dk.

[20] Coloured Petri Nets at the University of Aarhus. www.daimi.au.dk/CPnets.

[21] Pervasive Healthcare. www.healthcare.pervasive.dk.

[22] Radio Frequency Identification. www.rfid.org.

[23] Systematic Software Engineering A/S. www.systematic.dk.

Inheritance of Dynamic Behavior in UML

W.M.P. van der Aalst

Eindhoven University of Technology, Faculty of Technology and Management, Department of Information and Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
w.m.p.v.d.aalst@tm.tue.nl

Abstract. One of the key issues of object-oriented modeling and design is inheritance. It allows for the definition of subclasses that inherit features of some superclass. Inheritance is well defined for static properties of classes such as attributes and operations. However, there is no general agreement on the meaning of inheritance when considering the dynamic behavior of objects, captured by their life cycles. This paper studies inheritance of behavior in the context of UML. This work is based on a theoretical framework which has been applied and tested in both a process-algebraic setting (ACP) and a Petri-net setting (WF-nets). In this framework, four inheritance rules are defined that can be used to construct subclasses from (super-)classes. These rules and corresponding techniques and tools are applied to UML activity diagrams, UML statechart diagrams, and UML sequence diagrams. It turns out that the combination of blocking and hiding actions captures a number of important patterns for constructing behavioral subclasses, namely choice, sequential composition, parallel composition, and iteration. Both practical insights and a firm theoretical foundation show that our framework can be used as a stepping-stone for extending UML with inheritance of behavior.

1 Introduction

The Unified Modeling Language (UML) [11, 21] has been accepted throughout the software industry as the standard object-oriented framework for specifying, constructing, visualizing, and documenting software-intensive systems. One of the main goals of object-oriented design is the *reuse* of system components. A key concept to achieve this goal is the concept of *inheritance*. The inheritance mechanism allows the designer to specify a class, the *subclass*, that inherits features of some other class, its *superclass*. Thus, it is possible to specify that the subclass has the same features as the superclass, but that in addition it may have some other features.

The concept of inheritance is usually well defined for the *static structure* of a class consisting of the set of operations (methods) and the attributes. However, as mentioned, a class should also describe the dynamic behavior of an object. We will use the term "object life cycle" to refer to this behavior. The current version of UML, Version 1.4 [11], supports nine types of diagrams: class diagrams, object diagrams, use case diagrams, sequence diagrams, collaboration diagrams, statechart diagrams, activity diagrams, component diagrams, and deployment diagrams. Four of these types of diagrams, namely sequence diagrams, collaboration diagrams, statechart diagrams, and activity diagrams capture (part of) the behavior of the modeled system. Sequence diagrams and collaboration diagrams typically only model examples of interactions between objects (scenarios). Activity diagrams emphasize the flow of control from activity

to activity, whereas statechart diagrams emphasize the potential states and the transitions among those states. Both statechart diagrams and activity diagrams can be used to specify the dynamics of various aspects of a system ranging from the life cycle of a single object to complex interactions between societies of objects. Activity diagrams typically address the dynamics of the whole system including interactions between objects. Statechart diagrams are typically used to model an object's life cycle. Note that UML joins and/or is inspired by the earlier work on Message Sequence Diagrams [20] (for sequence diagrams), Statecharts [13] (for statechart diagrams), and Petri nets [19] (for activity diagrams).

Looking at the informal definition of inheritance in UML, it states the following: "The mechanism by which more specific elements incorporate structure and behavior defined by more general elements." [21]. However, only the class diagrams, describing purely structural aspects of a class, are equipped with a concrete notion of inheritance. It is implicitly assumed that the behavior of the objects of a subclass is an extension of the behavior of the objects of its superclass. Clearly, this is not sufficient to realize the full potential of inheritance [8, 14, 22, 23]. Therefore, our ultimate quest is to extend each diagram type of UML with suitable notions of inheritance. For this purpose we use theoretical results presented in [1–3, 5, 6] as a stepping stone. These results provide four notions of behavioral inheritance, inheritance preserving transformation rules, transfer rules, and advanced notions such as the Greatest Common Divisor (GCD) of a set of behavioral models.

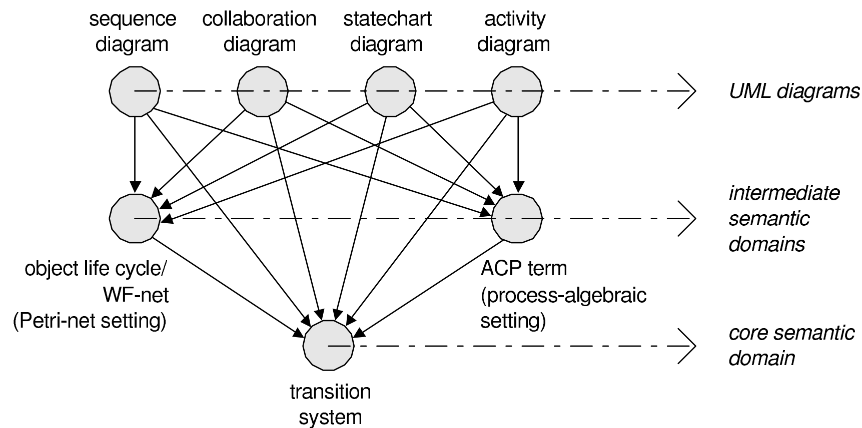


Fig. 1. Mapping UML behavior diagrams onto three semantic domains.

In this paper we follow the approach proposed in [8], i.e., instead of trying to give full formal semantics for UML we focus on selected parts of UML and map these parts onto a so-called *semantic domain*. A semantic domain is some formal language allowing for a precise definition of inheritance and equipped with analysis techniques to verify whether one process is a subclass of another process. Based on the theoretical

results presented in [1–3, 5, 6] we can use three semantic domains. This is illustrated in Figure 1. The *core semantic domain* is formed by transition systems using branching bisimilarity as an equivalence relation [6]. The mapping from specific models in both a Petri-net and process-algebraic setting to transition systems is given in [1–3, 5, 6]. In this paper, we explore the mapping from UML behavior diagrams to these semantic domains in order to incorporate inheritance of behavior in UML. A direct mapping from sequence, collaboration, statechart, and activity diagrams to the core semantic domain (i.e., transition systems) allows for full flexibility. An indirect mapping through one of the *intermediate semantic domains* (i.e., WF-nets and object life cycles in the Petri-net setting and ACP terms in the process-algebraic setting) allows for powerful analysis techniques, cf. the structure theory of Petri nets (e.g., invariants) and the equational theory of ACP. Moreover, the UML behavior diagrams are closely related to the two intermediate semantic domains. Consider for example the relation between activity diagrams and Petri nets.

Note that it is not our goal to provide a precise semantics for UML. This topic is relevant and has be debated many times before [7, 9], but is outside the scope of this paper.

The remainder of this paper is organized as follows. First, the theoretical results presented in [1–3, 5, 6] are introduced. Then, the four notions of inheritance are applied to sequence diagrams (Section 3), statechart diagrams (Section 4), and activity diagrams (Section 5). To conclude, we provide pointers to related work and summarize the main results.

2 Inheritance of behaviour

The goal of this section is to introduce four notions of inheritance and highlight some of the results theoretical results presented in [1–3, 5, 6]. Some of these results have been developed in a Petri-net setting (cf. [1–3, 5, 6]) others have been developed in a process-algebraic setting (cf. [5, 6]). However, the main ideas are generic and can be applied to any of the behavior diagrams in UML (i.e., sequence diagrams, activity diagrams, collaboration diagrams, and statechart diagrams). Therefore, we present the intuition behind our results and demonstrate their applicability to sequence diagrams, activity diagrams, and statechart diagrams.

2.1 An informal introduction to the four notions of inheritance

Diagrams such as sequence diagrams, activity diagrams, and statechart diagrams specify behavior of an object or system. The most elementary way of modeling behavior is the so-called *labeled transition system*. We consider this to be the core semantic domain (cf. Figure 1). A labeled transition system is a set of *states* plus a *transition relation* on states. Each transition is labeled with an *action*. Figure 2 shows a simple transition system representing an order processing system with five states (*s1*, *s2*, *s3*, *s4*, and *s5*). There are five transitions each labeled with a different action. The transition labeled with *accept_order* moves the system from the state *s1* to the state *s2*. For simplicity we assume that each transition system has one initial state (e.g., *s1* in Figure 2) and one

final state (e.g., *s5* in Figure 2). Note that any transition system with multiple initial and/or final states can be transformed in a transition system with one initial and one final state.

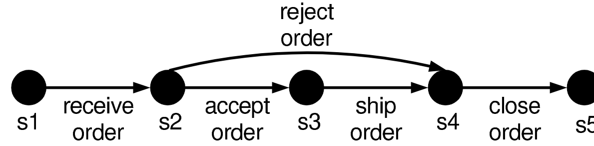


Fig. 2. A labeled transition system specifying an order processing system (TS_1).

States, transitions, and actions of a transition system should not be confused with the meaning of these notions in UML. In the context of this paper, the interpretation of these concepts depends on the UML diagrams being considered. For example, in a UML statechart diagram with a composite state consisting of multiple concurrent substates the composite state may correspond to many states in the corresponding labeled transition system (all possible combinations of states of the concurrent substates). In a UML sequence diagram, sending a message (i.e., a stimulus) corresponds to the execution of transition having the appropriate action label. In a UML activity diagram, there are action states which correspond to actions in the corresponding labeled transition system. In this section, we use the terms state, transition, and action in the context of a labeled transition system. In subsequent sections we will put these concepts in the context of UML diagrams specifying behavior.

We distinguish between *visible* actions and *invisible* actions. For comparing the behavior of two labeled transition systems only the visible actions are considered. The distinction between visible and invisible actions is fairly standard in process theory (see [6, 10] for pointers). Since it is not possible to distinguish between individual invisible actions (also referred to as silent actions), these actions are labeled τ. Two labeled transition systems are considered equivalent if their observable behaviors coincide, i.e., after abstracting from τ-actions one cannot detect any differences. From a formal point of view, branching bisimulation [6, 10] is used as an equivalence relation. However, we will avoid getting into formal definitions. Instead we refer to [1–3, 5, 6] for details.

In this paper, we focus on inheritance of dynamic behavior. Translated to labeled transition systems this translates to the following question: *When is one labeled transition system a subclass of another labeled transition system?* There seem to be many possible answers to this question. It is important to note that we have to ask this question from the viewpoint of the *environment*.

Assume that p and q are two labeled transition systems. The first answer is as follows.

> *If it is not possible to distinguish the external behavior of p and q when only actions of p that are also present in q are executed, then p is a subclass of q.*

Intuitively, this basic form of inheritance conforms to *blocking* actions new in p. In the remainder, labeled transition system p is said to inherit the *protocol* of q; the resulting

fundamental form of inheritance is referred to as *protocol inheritance*. Note that protocol inheritance specifies a *lower bound* for the behavior offered, i.e., any sequence of actions invocable on the superclass can be invoked on the subclass. Therefore, it is sometimes also referred to as “invocation consistency” [8, 22, 23].

The second answer to the above question is as follows.

> *If it is not possible to distinguish the external behavior of p and q when arbitrary actions of p are executed, but when only the effects of actions that are also present in q are considered, then p is a subclass of q.*

This second basic form of inheritance of behavior conforms to *hiding* the effect of actions new in p. Transition system p inherits the *projection* of transition system p onto the actions of q; the resulting form of inheritance is called *projection inheritance*. Note that projection inheritance can be considered as an *upper bound* for the behavior offered, i.e., any sequence of actions observable from the subclass should correspond to an observable sequence of the superclass (after abstraction). Therefore, it is sometimes also referred to as “observation consistency” [8, 22, 23].

To illustrate these two basic notions of inheritance we consider the labeled transition system specifying an order processing system shown in Figure 2. Suppose that this is the superclass named TS_1. Figure 3 shows another labeled transition system named TS_2. TS_2 is a subclass with respect to protocol inheritance because if we block the new actions, the observable behavior of TS_2 coincides with the observable behavior TS_1. If action *subcontract_order* is never executed, the original behavior is preserved. Note that in Figure 3 and subsequent figures new actions (i.e., action not appearing in Figure 2) are highlighted. Also note that TS_2 is *not* a subclass of TS_1 with respect to projection inheritance. If we hide the two new actions, there is an occurrence sequence where *receive_order* is directly followed by *close_order* without executing *reject_order* or *accept_order* and *ship_order* in-between. Clearly this behavior is not possible in TS_1.

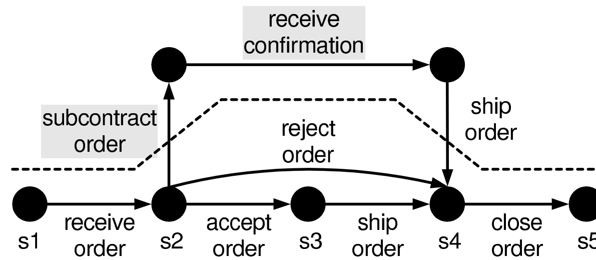


Fig. 3. A subclass with respect to protocol inheritance (TS_2).

Labeled transition system TS_3 shown in Figure 4 is a subclass with respect to projection inheritance. The new action *send_invoice* is executed in parallel with *ship_order*. (We are assuming interleaving semantics here.) If *send_invoice* is renamed to τ, then action *accept_order* is always followed by *ship_order* which in turn is followed by

close_order. Therefore, the observable behavior of TS_3 coincides with the observable behavior of TS_1 after abstracting from *send_invoice*, i.e., TS_3 is a subclass of TS_1 with respect to projection inheritance. Note that TS_3 is not a subclass of TS_1 with respect to protocol inheritance. If *send_invoice* is blocked, the process gets stuck after executing *ship_order*.

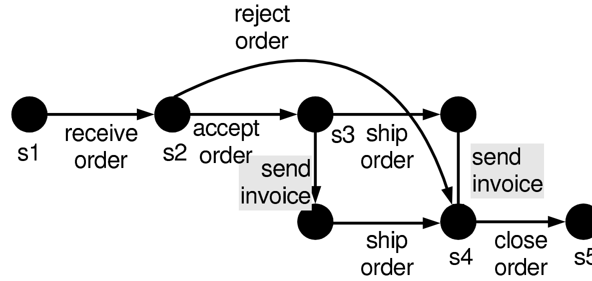


Fig. 4. A subclass with respect to projection inheritance (TS_3).

There are essentially two ways to combine the two basic notions of inheritance into stronger or weaker notions of inheritance. Protocol/projection inheritance is the most restrictive form of inheritance which combines both basic notions at the same time. If p is a subclass of q with respect to protocol/projection inheritance, then p is a subclass of q with respect to protocol inheritance *and* projection inheritance. Life-cycle inheritance is the most liberal form of inheritance: The set of new actions is partitioned into hidden and blocked such that the observable behavior of the subclass equals the behavior of the superclass. Note that protocol and/or projection inheritance implies life-cycle inheritance.

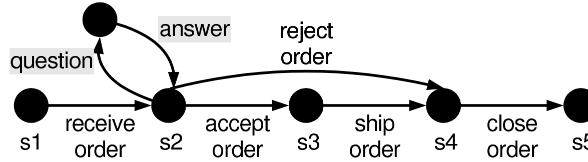


Fig. 5. A subclass with respect to protocol/projection inheritance (TS_4).

TS_4 shown in Figure 5 is a subclass of TS_1 with respect to protocol/projection inheritance because either blocking or hiding the two new actions results the observable behavior of TS_1. If action *question* is blocked, the new behavior is never activated. If actions *question* and *answer* are renamed to τ, their presence cannot be observed.

Labeled transition system TS_5 shown in Figure 6 is a subclass of TS_1 with respect to life-cycle inheritance. By blocking the new action *subcontract_order* and hiding action *send_invoice* the resulting observable behavior coincides with TS_1. Note that TS_5

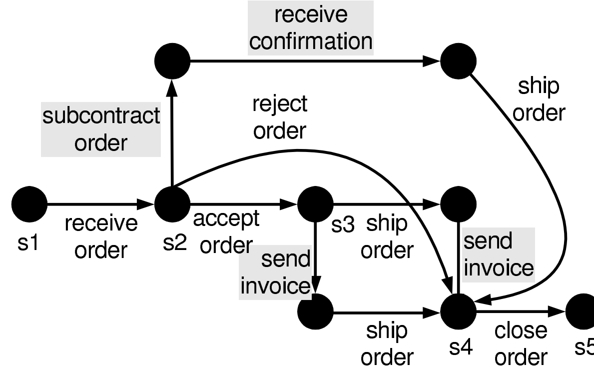


Fig. 6. A subclass with respect to life-cycle inheritance (TS_5).

is not a subclass of TS_1 with respect to any of the other three notions of inheritance. All the other notions either block and/or hide all the new actions while for life-cycle inheritance each individual new action is either blocked or hidden. Note that TS_5 is also a subclass of TS_2 with respect to projection inheritance. Moreover, TS_5 is a subclass of TS_3 with respect to protocol inheritance.

To summarize, we have identified four notions of inheritance based on two fundamental mechanisms: hiding and blocking. The most restrictive notion of inheritance is protocol/projection inheritance. Of the four transition systems TS_2, TS_3, TS_4, and TS_5 only TS_4 is a subclass of TS_1 with respect to protocol/projection inheritance. Life-cycle inheritance is the most liberal form of inheritance and each of the four transition systems TS_2, TS_3, TS_4, and TS_5 is a subclass of TS_1 with respect to life-cycle inheritance.

2.2 Inheritance preserving transformation rules and other results

Based on the four notions of inheritance, we have developed a set of tools (e.g., Woflan [24]) and obtained powerful theoretical results. These theoretical results have been presented in [1–3, 5, 6] and in this section we only highlight some of them.

In both a Petri-net and a process-algebraic setting we have developed a comprehensive set of *inheritance preserving transformation rules* (cf. Table 1). A detailed description of these rules is beyond the scope of this paper. Therefore, we give an informal description of four inheritance preserving transformation rules: PP, PT, PJ, and PJ3. Transformation rule PP is used to add loops such that each loop eventually returns to the state where it was initiated. If these loops contain only new or invisible actions, protocol/projection inheritance, and therefore also the other three notions of inheritance, are preserved. TS_4 can be constructed from TS_1 using this rule, and therefore, it automatically follows that TS_4 is a subclass of TS_1 under protocol/projection inheritance. PT preserves protocol inheritance and adds alternative behavior. TS_2 can be constructed from TS_1 using this rule. PJ and PJ3 both preserve projection inheritance. PJ can be used to insert new actions in-between existing actions. PJ3 can be used to add parallel

behavior. TS_3 can be constructed from TS_1 using rule PJ3. The four rules (PP, PT, PJ, and PJ3) correspond to design constructs that are often used in practice, namely iteration, choice, sequential composition, and parallel composition. If the designer sticks to these rules, inheritance is guaranteed. It should be noted that the precise formulation of these rules depends of the modeling language being used and is not as straightforward as it may seem (e.g., the added parts should not introduce deadlocks and terminate properly). The four inheritance preserving transformation rules have been formulated in terms of object life cycles and WF-nets (Petri-net-based modeling languages, cf. [1–3, 5, 6]) but also in terms of ACP (a process-algebraic language, cf. [5, 6]). Moreover, in the process-algebraic setting additional rules (PJ2, LC1, LC2, and LC3) have been formulated [5, 6].

name	adds	preserves
PP	loops containing new behavior	all notions of inheritance
PT	new alternatives starting with a new action	only protocol and life-cycle inheritance
PJ	new actions inserted in-between existing ones	only projection and life-cycle inheritance
PJ3	new actions in parallel with existing ones	only projection and life-cycle inheritance

Table 1. Overview of inheritance preserving transformation rules.

Based on the inheritance preserving transformation rules we have also developed a comprehensive set of *transfer rules*. These transfer rules can be used to migrate instances from a subclass to a superclass and vice versa. Suppose that p is a subclass of q constructed using the rules PP, PT, PJ, and PJ3. For any state in p it is possible to transfer an instance (e.g., an object of a class whose life-cycle is specified by transition system p) to q such that the transfer is instantaneous (i.e., no postponements needed) and does not introduce syntactic errors (e.g., deadlocks, livelocks, and improper termination) nor semantic errors (e.g., the double execution of actions or unnecessary skipping of actions). Moreover, it is also possible to transfer instances from subclass p to superclass q without any problems. Note that the transfer rules are derived from the transformation rules introduced earlier. The transfer rules to move a case to a subclass are: r_{PT}, r_{PP}, r_{PJ}, $r_{PJ3,C}$ and $r_{PJ3,P}$. The transfer rules to move a case to a superclass are: $r^{-1}_{PT,C}$, $r^{-1}_{PT,P}$, r^{-1}_{PP}, r^{-1}_{PJ}, and r^{-1}_{PJ3}. See [3] for the specification of these transfer rules in a Petri-net setting.

Each of the four inheritance relations provides an ordering on labeled transition systems and can be used to define concepts such as the GCD (Greatest Common Divisor) of two processes. The concept of GCD was introduced in [3] and a detailed analysis of this concept is given in [2]. Since none of the inheritance relations is a lattice, there is a trade-off between "uniqueness" and "existence". By using a weak notion of GCD, existence is guaranteed but there may be multiple GCD's. By using a stronger notion, existence is no longer guaranteed but if the GCD exists, it is unique. Given this tradeoff, we define the notion of Maximal Common Divisor (MCD). An MCD is a "smallest" superclass of both p and q under life-cycle inheritance. If a set of labeled transition systems is related under inheritance via subclass-superclass relationships, it is generally

quite easy to find the GCD. If this is not the case, the computation of a GCD is more involved and there are typically multiple candidates (i.e., MCD's). Similarly results hold for LCM's (Least Common Multiple) and MCM's (Minimal Common Multiple) [2, 3].

These results illustrate the strong theoretical foundation for inheritance of dynamic behavior. Unfortunately, its application has been limited to the workflow domain [3]. In this paper, we want to demonstrate that these results can also be used as a stepping stone for extending the behavior diagrams in UML (i.e., sequence diagrams, activity diagrams, collaboration diagrams, and statechart diagrams) with inheritance.

2.3 Checking inheritance using Wolfan

To illustrate the applicability of the four inheritance concepts we refer to our workflow analysis tool Woflan [24]. Woflan is based on Petri-nets and aims at the verification of workflow processes. Besides checking for deadlocks and other design errors, Woflan also supports the four notions of inheritance. Given two workflow models, Woflan is able to check whether one model is a subclass of the other model. The current version assumes that these models are expressed in terms of Petri nets. However, we have developed translations from concrete systems such as COSA (COSA Solutions/Thiel AG), Staffware (Staffware PLC), and Protos (Pallas Athena) and alternative modeling techniques such as workflow graphs and event-driven process chains [3]. Moreover, the inheritance-checker is relatively independent of the modeling language used and the results are not restricted to workflow processes but apply to any behavioral model. These practical results demonstrate the practical potential of the results presented in [1–3, 5, 6] in the context of UML.

3 Sequence diagrams

The first diagram type we consider is the UML sequence diagram. A sequence diagram has two dimensions: (1) the vertical dimension represents time and (2) the horizontal dimension represents different instances. Sequence diagrams are typically used to describe specific scenarios of interaction among objects. Although UML allows for variations such as iteration, conditional, and timed behavior, we assume that the sequence diagram is restricted to lifelines, messages (i.e., communications of type procedure call, asynchronous, and return), activation, and concurrent branching. Under these assumptions it is fairly straightforward to map a sequence diagram onto an labeled transition system (core semantic domain) or a Petri net (intermediate semantic domain) [16, 20]. In fact, under these assumptions any sequence diagram corresponds to a so-called marked graph [19], i.e., a Petri net where places cannot have multiple inputs/outputs. This observation indicates that only projection inheritance is relevant for this diagram type. (If there are no choices, it makes no sense to block behavior since this will only cause deadlocks.) As a result, only the projection-inheritance preserving transformation rules are relevant.

Consider the two sequence diagrams shown Figure 7. The right-hand side diagram is a subclass of the left-hand side diagram under projection inheritance. There are two ways to verify this. First of all, we can map both sequence diagrams onto any of the

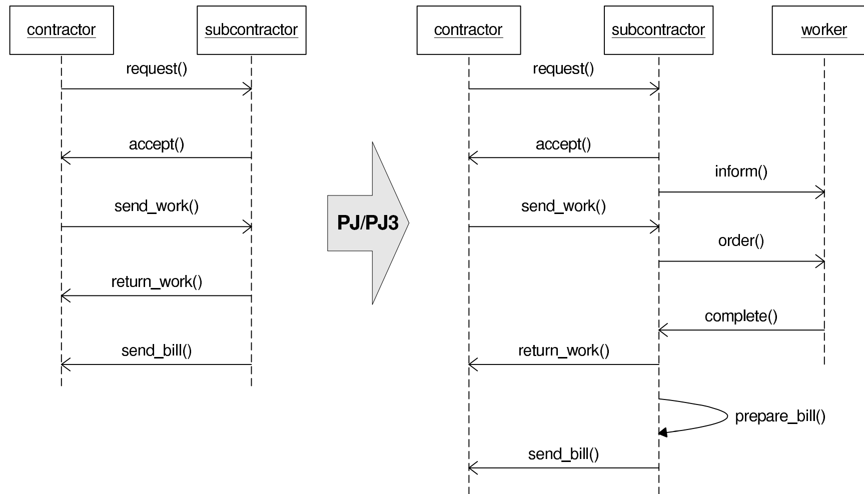


Fig. 7. A subclass sequence diagram constructed using rules PJ and PJ3.

semantics domains shown in Figure 1 and then check inheritance in the corresponding domain. Second, one can apply the projection-inheritance preserving transformation rules PJ and PJ3 described in Section 2.2. PJ can be used to add action *prepare_bill* and PJ3 can be used to add the lifeline *worker* and the actions *inform*, *order*, and *complete*. Note that in Figure 7 we assumed a one-to-one correspondence between messages and actions. Moreover, we did not use communications of type procedure call, activation, and concurrent branching. However, translation of PJ and PJ3 to sequence diagrams can easily deal with these concepts.

Collaboration diagrams are closely related to sequence diagrams. In essence they provide a different view on the identical structures [21]. Therefore, the results obtained for inheritance of sequence diagrams can easily be transferred to collaboration diagrams.

4 Statechart diagrams

In contrast to sequence diagrams, statechart diagrams are typically not used to specify scenarios. Instead they are used to model the life cycle of object. Since the 1987 paper by David Harel [13] there has been an ongoing discussion on the semantics of statecharts. Clearly, any of these semantics can be mapped onto our core semantic domain (transition systems) and thus implicitly use the four notions of inheritance described in Section 2.1. Moreover, in [8] a partial mapping onto CSP is given (we can use a similar semantic mapping to ACP) and in [22, 23] a mapping onto object behavior diagrams is given. Instead of providing yet another mapping of statechart diagrams onto some semantic domain we provide two examples to demonstrate our inheritance notions and the related transformation rules.

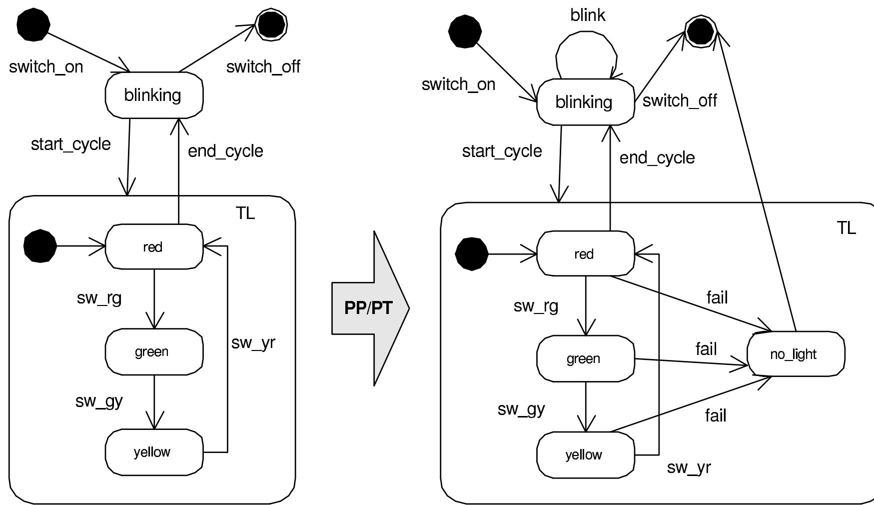


Fig. 8. A subclass statechart diagram constructed using rules PP and PT.

Figure 8 shows two statechart diagrams. The right-hand side diagram is a subclass of the left-hand side diagram under protocol inheritance. Any sequence of actions possible in the left-hand side diagram is also possible in the right-hand side diagram. The superclass statechart diagram models a Dutch traffic light which is either in state *blinking* or in composite state *TL*. The composite state is decomposed in three substates: *red*, *green*, and *yellow*. The subclass statechart diagram extends the superclass in two ways. First of all, the self transition *blink* is added. Second, the composite state is extended to allow for a traffic light which fails. State *no_light* corresponds to a malfunctioning traffic light which is shut down. The extension involving transition *blink* can be realized by applying PP, this is the protocol/projection inheritance preserving transformation rule which introduces loops which can be blocked or hidden. The extension involving state *no_light* can be realized by applying PT, this is the protocol inheritance preserving transformation rule which introduces alternatives which can be blocked.

Another example illustrating the application of our framework to statechart diagrams is given in Figure 9. The right-hand side diagram is a subclass of the left-hand side diagram under projection inheritance. The subclass statechart diagram (right) extends the superclass (left) in two ways. First of all, the traffic light has four phases instead of three including a state *red+yellow*. Second, the composite state *TL* has now two concurrent regions: one corresponding to the original traffic light with one additional phase and one corresponding to a mechanism to count the number of cars. If we abstract from this new mechanism and the additional phase, we obtain the original traffic light. Therefore, it is easy to verify that the right-hand side diagram is a subclass of the left-hand side diagram under projection inheritance. However, it is also possible to demonstrate this by applying the two projection inheritance preserving transformation

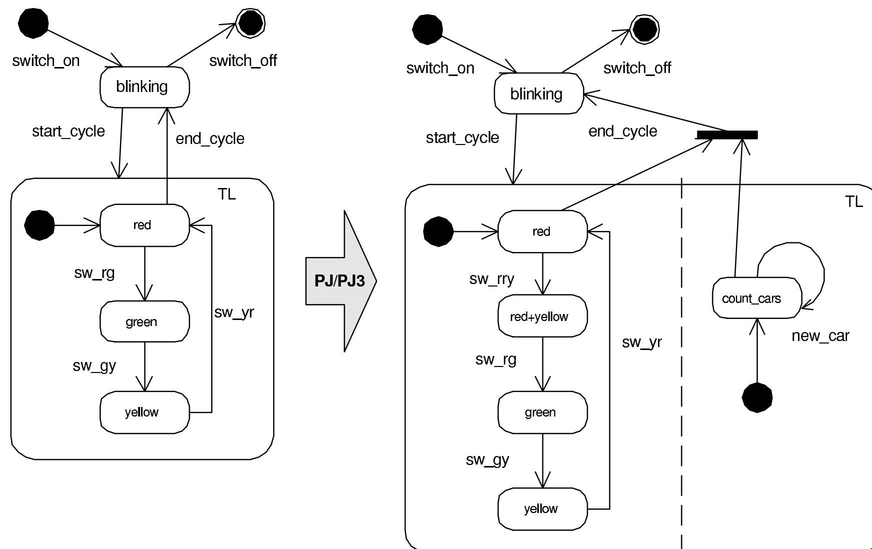


Fig. 9. A subclass statechart diagram constructed using rules PJ and PJ3.

rules PJ and PJ3 mentioned in Section 2.2. PJ can be used to insert the additional state *red+yellow* and PJ3 can be used to add the concurrent region for counting cars.

5 Activity diagrams

An activity diagram is a variation of a state machine in which the states represent the performance of actions or subactivities and the transitions are triggered by the completion of the actions or subactivities. Activity diagrams are typically used for modeling behavior which transcends the life cycle a single object. Therefore, it supports notations such swimlanes and is often used for workflow modeling. Compared to classical statecharts, activity diagrams allow for actions states, subactivity states, decisions and merges (both denoted by a diamond shape), object flows, and concurrent transitions (to model synchronization and forks). Note that swimlanes do not influence the behavior of an activity diagram. The semantics of activity diagrams is still under discussion. However, clearly many ideas have been adopted from Petri nets and in the proposal for UML 2.0 token passing is used as the main mechanism to specify the semantics of activity diagrams [12]. Therefore, we use WF-nets [3] as the semantic domain to map activity diagrams on. Concurrent transitions are mapped Petri-net τ transitions, decisions and merges are mapped onto places, actions states are mapped onto Petri-net transitions, object flows are mapped onto places, transitions are mapped onto τ transitions and places, etc.

Again we use an example to illustrate the application of our inheritance notions. Consider the two activity diagrams shown in Figure 10. The right-hand side diagram is a subclass of the left-hand side diagram under life-cycle inheritance. Note that the

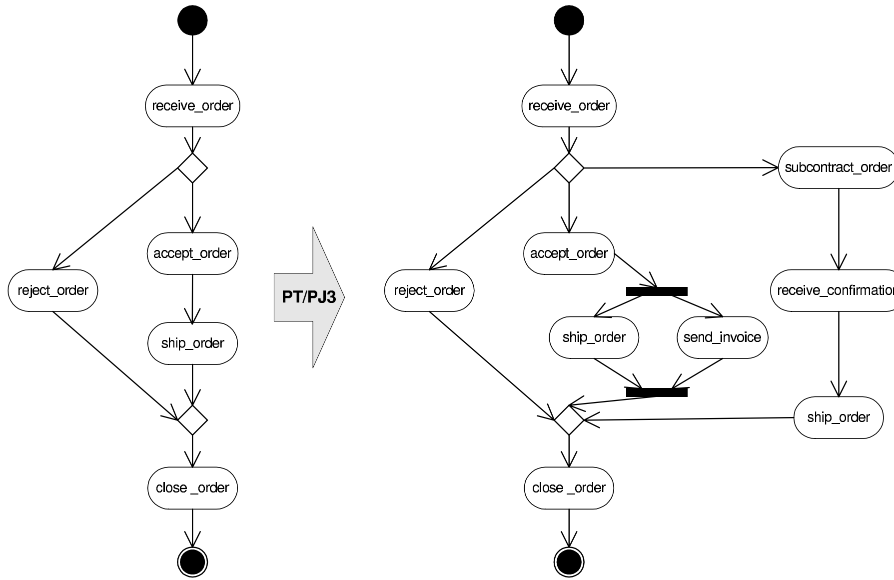


Fig. 10. A subclass activity diagram constructed using rules PT and PJ3.

right-hand side diagram is not a subclass under any of the other three notions of inheritance: Action *send_invoice* needs to be hidden while action *subcontract_order* needs to be blocked, therefore none of the other notions applies. Also note that left-hand side diagram corresponds to the labeled transition system shown in Figure 2. The right-hand side diagram corresponds to the labeled transition system shown in Figure 6. The subclass activity diagram (right) extends the superclass (left) in two ways: (1) *send_invoice* can be added using projection inheritance preserving transformation rule PJ3, and (2) the alternative sequence starting with *subcontract_order* can be added using protocol inheritance preserving transformation rule PT.

In Section 2.3 our verification tool Woflan [24] was already mentioned. Using a straightforward mapping of activity diagrams onto WF-nets, we can use Woflan to check whether the right-hand side activity diagram in Figure 10 is a subclass of the left-hand side diagram under life-cycle inheritance. As Figure 11 shows, this is indeed the case. The interested reader can download Woflan from http://www.tm.tue.nl/it/woflan.

6 Related work

The literature on object-oriented design and its theoretical foundations contains several studies related to the research described in this paper. In [26], abstraction in a process-algebraic setting is suggested as an inheritance relation for behavior. Other research on inheritance of behavior or related concepts such as behavioral subtyping are presented in [4, 15, 17, 18, 25]. The variety of inheritance relations reported in the literature is not surprising if one considers, for example, the large number of semantics that exist for concurrent systems (see, for example, [10]). For an elaborate overview of other

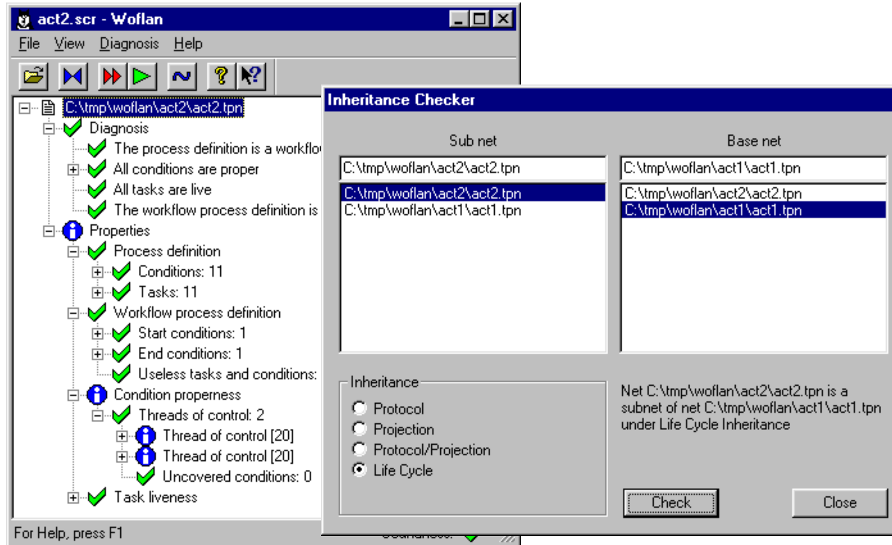


Fig. 11. Woflan shows that the right-hand side activity diagram in Figure 10 is indeed a subclass of the left-hand side diagram under life-cycle inheritance.

approaches we refer to [6]. Although many authors mention the need for inheritance of behavior in the context of UML, in most cases the application to concrete UML diagrams is missing. Consider for example the work presented in [14]. The authors provide a rigorous framework for behavioral inheritance, but do not "lift" the framework to the level of statechart or activity diagrams. Other authors focus specifically on inheritance of statecharts [8, 22, 23]. In [8] inheritance of statechart diagrams is investigated using CSP as a semantic domain. In [22, 23] Object/Behavior Diagrams are used as a semantic domain. It is encouraging to see that in [8, 22, 23] similar notions of inheritance are used: "invocation consistency" corresponds to our protocol inheritance and "observation consistency" corresponds to our projection inheritance. Note that we allow for two additional notions of inheritance (protocol/projection inheritance and life-cycle inheritance), provide inheritance preserving transformation and transfer rules, and also extend our work to sequence and activity diagrams.

7 Conclusion

In this paper, we investigated the applicability of the theoretical results on behavioral inheritance presented in [1–3, 5, 6] to the UML diagrams dealing with behavior. Although these theoretical results have been developed in the context of specific models for concurrency (i.e., Petri-nets and process algebra) there is a common core which has been illustrated using labeled transition systems (our core semantic domain). We have demonstrated that this core can be lifted to the level of concrete UML diagrams. In particular, we have applied the four inheritance notions and corresponding transformation

rules to sequence diagrams, activity diagrams, and statechart diagrams. In this paper, we used a rather pragmatic approach not aiming at the full expressive power of UML. The full UML standard simply contains too many features and is not defined (yet) sufficiently to allow for our ultimate quest: Defining inheritance for sequence diagrams, activity diagrams, and statechart diagrams as it is defined for object diagrams. However, the examples presented in this paper show that the theoretical results can be lifted to the level of concrete UML diagrams.

References

1. W.M.P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-net-based Approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, Berlin, 1997.
2. W.M.P. van der Aalst and T. Basten. Identifying Commonalities and Differences in Object Life Cycles using Behavioral Inheritance. In J.M. Colom and M. Koutny, editors, *Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 32–52. Springer-Verlag, Berlin, 2001.
3. W.M.P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270(1-2):125–203, 2002.
4. P. America. Designing an Object-Oriented Programming Language with Behavioral Subtyping. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Foundation of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, Berlin, 1991.
5. T. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, December 1998.
6. T. Basten and W.M.P. van der Aalst. Inheritance of Behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001.
7. R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Precise Semantics for Object-Oriented Modeling Techniques. In M. Aksit and S. Matsuoka, editors, *11th European Conference on Object-Oriented Programming ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 344–366. Springer-Verlag, Berlin, 1997.
8. G. Engels, R. Heckel, and J.M. Küster. Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In M. Gogella and C. Kobryn, editors, *4th International Conference on The Unified Modeling Language (UML 2001)*, volume 2185 of *Lecture Notes in Computer Science*, pages 272–286. Springer-Verlag, Berlin, 2001.
9. R. Eshuis and R. Wieringa. Comparing Petri Nets and Activity Diagram Variants for Workflow Modelling- A Quest for Reactive Petri Nets. In H. Ehrig, W. Reisig, and G. Rozenberg, editors, *Petri Net Technologies for Communication Based Systems*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 2002.
10. R.J. van Glabbeek. The Linear Time - Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves. In E. Best, editor, *Proceedings of CONCUR 1993*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer-Verlag, Berlin, 1993.
11. Object Management Group. *OMG Unified Modeling Language, Version 1.4*. OMG, http://www.omg.com/uml/, 2001.
12. Object Management Group. *OMG Unified Modeling Language 2.0 Proposal, Revised submission to OMG RFPs ad/00-09-01 and ad/00-09-02, Version 0.671*. OMG, http://www.omg.com/uml/, 2002.

13. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
14. D. Harel and O. Kupferman. On the Behavioral Inheritance of State-Based Objects. Technical Report MCS99-12, The Weizmann Institute of Science, Israel, 1999.
15. G. Kappel and M. Schrefl. Inheritance of Object Behavior - Consistent Extension of Object Life Cycles. In J. Eder and L.A. Kalinichenko, editors, *Proceedings of the Second International East/West Database Workshop (EWDW 1994)* , pages 289–300. Springer-Verlag, Berlin, 1995.
16. E. Kindler, A. Martens, and W. Reisig. Inter-Operability of Workflow Applications: Local Criteria for Global Soundness. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 235–253. Springer-Verlag, Berlin, 2000.
17. B. Liskov and J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
18. O. Nierstrasz. Regular Types for Active Objects. *ACM Sigplan Notices*, 28(10):1–15, October 1993. Special issue containing the proceedings of the 8th. annual conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA'93, Washington DC, 1993.
19. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.
20. E. Rudolph, J. Grabowski, and P. Graubmann. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.
21. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, Reading, MA, USA, 1998.
22. M. Schrefl and M. Stumptner. On the Design of Behavior Consistent Specialization of Object Life Cycles in OBD and UML. In M. Papazoglou, S. Spaccapietra, and Z. Tari, editors, *Advances in Object-Oriented Data Modelling*, pages 65–104. MIT Press, 2000.
23. M. Stumptner and M. Schrefl. Behavior Consistent Inheritance in UML. In Alberto H. F. Laender et al. editor, *Proceedings of the 19th International Conference on Conceptual Modeling (ER 2000)*, volume 1920 of *Lecture Notes in Computer Science*, pages 527–542. Springer-Verlag, Berlin, 2000.
24. H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.
25. H. Wehrheim. Subtyping Patterns for Active Objects. In H. Giese and S. Philippi, editors, *Proceedings 8ter Workshop des GI Arbeitskreises GROOM (Grundlagen objekt-orientierter Modellierung)*, volume 24/00, Münster, Germany, 2000. University of Münster.
26. R.J. Wieringa. *Algebraic Foundations for Dynamic Conceptual Models*. PhD thesis, Free University, Amsterdam, The Netherlands, 1990.

A Colored Petri Net for a Multi-Agent Application

Danny Weyns, Tom Holvoet

Computer Science Department, Katholieke Universiteit Leuven

Celestijnenlaan 200A, B-3001 Leuven, Belgium

http://www.cs.kuleuven.ac.be/~danny/home.html

{danny,tom}@cs.kuleuven.ac.be

Abstract

In this paper we present a Colored Petri Net (CPN) for a multi-agent application. In particular we modeled the Packet-World. In our research we use the packet-world as a case to study the fundamentals of agents' social behavior. Our approach is to combine experiments with conceptual modeling. We start from a very basic model and then add social skills in a modular way. Integrating new social skills by means of adding new modules offers us a clear conceptual view on the evolution of agents and the environment. With a conceptual view we mean: (i) which concepts does an agent need in order to acquire a new kind of social ability, (ii) which infrastructure is necessary in the environment to support these abilities, (iii) how do these concepts relate to each other? With the insights we learn from the case study, we gradually develop a generic conceptual model for social agents situated in a MAS. In this paper we first present a CPN for a basic model of the packet-world. This model consists of agents that can only interact through passive objects in the environment. Because interaction is the central issue of multi-agent systems, we have incorporated basic infrastructure for agent coordination straight away into our basic model. Then we extend the model, making it possible for the agents to communicate information with each other. Communication is the basis for social organization. Besides the concrete realization of a CPN for a multi-agent application, the model we present in this paper has the potential to support our future research of agents' social behavior. Our major motives for using CPNs as modeling tool are (i) CPNs gives a clear conceptual view on agents and the environment wherein they live, and (ii) CPNs support neat verification and formalization.

Keywords

Colored Petri Nets, Multi-Agent Systems

1. Introduction

In this introduction we first situate our research goals in the domain of Multi-Agent Systems (MAS). Next we explain how we tackle the problems we want to solve and motivate our choice for Colored Petri Nets (CPN) as a tool for modeling MASs. Then we situate the subject of this paper in our research. We conclude with an overview of the paper.

1.1 Fundamentals of Sociality in MASs

The importance of MASs as a design concept for today's software is beyond dispute. A MAS models a part of the world as a community of autonomous agents that interact in an environment. In a MAS the activity is distributed over the agents of the community. The intelligence of the systems comes from the interaction between the agents, rather then from their individual capabilities. This contrasts to the approach of the classical artificial intelligence where an agent acts as an independent "cognitive reasoning machine".

A MAS is a society of agents that live and work together. Living in a community requires a number of social skills. Until now the agent research community has paid little attention to the fundamentals of sociality in MASs. Many unanswered questions remain. The lack of insight in agents' sociality limits the potential of the agent-based approach. We quote N. Jennings in [6]:

> *"To realize the full potential [of MASs], a better understanding is needed of the impact of sociality [..] on an individuals behavior and of the link between the behavior of the individual agents and that of the overall system."*

Until now, research about the fundamentals of agents' sociality can be divided into two approaches. In the first approach research is mainly experiment-oriented. Some references are [8][9][10][11][12]. These projects explore new kind of interactions and rules for setting up social structures. What we can see is that from the interaction of the agents new functionality's emerge that go beyond the sum of the capabilities of the individuals. The second approach intends to conceptualize the social aspects of agents in a MAS. Some examples are [6][13][14][15]. One group of researchers has integrated certain aspects of agents' sociality in a formal model of an agent (e.g. the BDI-model). Another group has set out some thoughts how agents' sociality and the organization of a MAS might be structured.

We conclude that most of the work that has been done so far, has one or more of the following characteristics:

- the research focuses on one particular aspect of sociality in MAS
- the research starts from a particular point of view
- most of the research is done separately from one another
- the focus is mainly directed on *how* social behavior could emerge in a MAS, much less attention is devoted to the questions *why* and *when* social behavior arises

1.2 The goals of our research

The goal of our research is to get a better understanding of sociality in MASs. Therefore we intend to build a *generic conceptual model* of social agents situated in a MAS. Our approach is to combine experiments with conceptual modeling, the two approaches we mentioned in the previous section. We use a case application to explore different kind of social behaviors. Parallel with experiments we build a conceptual model. We start from a very basic model and then add social skills in a modular way. Integrating new social skills by means of adding new modules offers us a clear conceptual view on the evolution of agents and the environment. With a conceptual view we mean: (i) which concepts does an agent need in order to acquire a new kind of social ability, (ii) which infrastructure is

necessary in the environment to support these abilities, (iii) how do these concepts relate to each other? With the insights we learn from the case study, we gradually will develop a generic conceptual model for social agents situated in a MAS. Therefore we have to generalize the insights we learned from the case application in order to build abstract models for different classes of social skills.

1.3 Modeling MASs with Colored Petri Nets

When we set out our approach the question arises how we should model the case application. Since Petri nets [1] have a long tradition to describe and analyze concurrent processes, they where excellent candidates. Colored Petri Nets (CPN) [2] combine the best of classical Petri nets and high level programming languages, and are for that very popular. CPNs have an intuitive graphical representation that paves the way for clear conceptual modeling of complex systems. The behavior of a system modeled with a CPN can be analyzed, not only by means of a simulation but also on a formal base. It is remarkable that CPNs, which offer most of the ingredients to tackle the complexity of multi-agent systems, are little used to model and study them. Some interesting references are [16][17][18][19][20][21][22]. The most far-reaching use of CPNs for modeling MASs is from Ferber. Ferber developed a formalism called “Basic Representation of Interactive Components” (BRIC). BRIC is based on a component approach, each of the primitive components (“bricks”) described with a CPN. In his standard work “Multi-Agent Systems” [3], Ferber proposes an extensive set of BRIC components, each of them representing a generic model for a specific part of a MAS. Inspired by his ideas we decided to use CPNs in our research. In contrast with Ferber, who uses CPNs for an *operative representation of the functioning of a MAS* we use CPNs for a *conceptual modeling of sociality in MASs*.

1.4 Situating the paper in our research, overview of the paper

The multi-agent application we use in our research is that of the *Packet-World*. Originally, Huhns and Stephens proposed this application in [7] as a research topic to investigate sociality in MASs. The packet-world consists of a number of different colored packets that are scattered over a rectangular grid. Agents that live in this virtual world have to collect those packets and bring them to their corresponding colored destination. The agents have only a limited view on the world. The packet-world offers a rich set of fundamental characteristics for a broad range of multi-agent systems. E.g., agents may perform better their job when they share their information or when they set up a form of cooperation. In this paper we describe two models of the packet-world. These two models form a solid basis for our future research of agents’ social behavior. After an intuitive description of the packet-world, in section 3 we present a CPN for a basic model. This model consists of agents that can only interact through passive objects in the environment. Next in section 4, we extend the model, making it possible for the agents to communicate information with each other. Communication is the basis of social organization. In section 5 we give results of our first experiments with the two models. Finally, we conclude and look to future work in section 6.

The CPNs that we present in this paper are designed with the Design/CPN tool [4][5]. In order to keep a clear view on the models, we limit the number of agents to two.

2. The packet-world

2.1 Introduction

Consider a rectangular grid of size S. The grid contains a number of colored packets and agents. It is the agents’ job to collect the packets and bring them to their corresponding colored destinations. The grid contains one destination for each color. Figure 1 shows an example of a packet-world of size 8 with 3 agents.

In the packet-world agents can interact with the environment in a number of ways. We allow agents to perform a number of basic interactions with the passive objects of the environment. First, an agent

can make a step to one of the free neighbor fields around him. Second, if an agent is not carrying any packet, he can pick up one from one of his neighbor fields. Third, an agent can put down the packet he carries on one of the free neighbor fields around him or of course on the destination field of that particular packet. Finally an agent may wait for a while and do nothing.

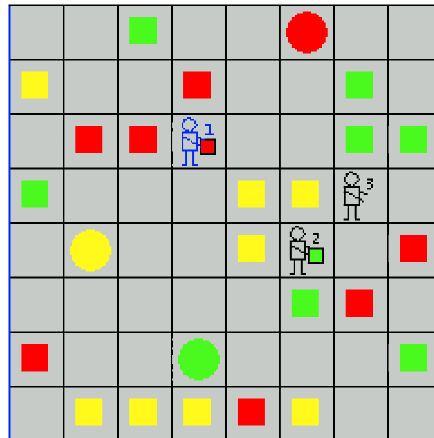


Figure 1. The packet-world (squares are packets, circles are delivering points)

It is important to notice that each agent has only a limited view on the world. This view covers only a small part of the environment around the agent. This property of limited knowledge is typical for the agents of a multi-agent system. In our model, the *view-size* of the world expresses how far (i.e. how many squares) an agent can "see" around him.

In our model we use a simple measure to indicate how efficient the agents perform their job. Each time an agent makes a step or moves a packet (by picking it up, putting it down or step with it) a counter is incremented. At each point in time the value of this counter indicates how much energy the agents have invested in their work so far.

In the basic model for the packet-world we limit the agents possible interactions with the environment to the basic set we mentioned above. We modeled the basic agents without any social skills. Their *goal* is to collect the packets of the world and bring them to their destinations. This general goal can be divided into a set of *primary goals*. In short, those agents act in a repeating cycle driven by two primary goals: look for a packet and pick it up, look for the destination and deliver the packet.

In the extended model, agents can interact with each other. This interaction is the foundation of cooperation between the agents. For the packet-world one can imagine different kinds of cooperation. Agents can for example agree on a plan to form a chain and pass on packets to each other. We modeled another form of cooperation. In the extended model we present in this paper, we integrated facilities into the basic model to let agents communicate with each other. In particular, agents are extended with functionality to request information from each other. Instead of exploring the world to find a target an agent does not see, the agent now can ask a visible colleague for the desired information. If the requested agent knows or sees the asked information he can respond the query with the information. This allows the requesting agent to act more efficient.

2.2 Actions, influences and reactions

Agents of a multi-agent system are endowed with autonomy. They are driven by a set of goals. In order to achieve those goals agents undertake *actions*. When an agent acts in the environment, e.g. he picks up a packet next to him he has no full guarantee that this action will succeed. Another agent might be trying to pick up the same packet at about the same time. As a consequence only one of

them will get the packet leaving the other with empty hands. Therefore we say that an action of an agent results in an ***influence*** in the environment. Influences result in ***reactions*** from the environment. Each influence can succeed or fail. For example if an agent performs the action "pick" he invokes the influence "perform pick" on the environment. If the action succeeds, the environment reacts with the reaction "do pick", if the action fails the reaction will be "can't pick"**.** So it is only after the reaction of the environment to all the performed influences (at about the same point of time) that the agents actually experience the result of their intended actions. When an agent is notified about the result of the action he undertook, we say that the agent ***consumes*** the result of his produced influence.

2.3 Agent state

An agent can only decide to perform an action if he is endowed with some attitudes and has some information at his deposal. In section 2.1, we mentioned an agent is driven by a set of primary goals. Agents act to achieve those goals. Therefore they perform influences into the environment, as described in section 2.2.

When an agent selects an action, he has to take the state of the world into account. If an agent for example decides to pick up a packet, first of all, he must be aware of the fact he actually does not carry a packet. In general this means the agent must possess some state of his own. In our model of the packet-world an agent maintains the state of his position and whether he actually carries a packet or not. Further, the agent must "see" the packet near to him. He needs some information about the environment around him in order to act. We call this information the ***view*** of the agent. In our model, regularly each agent gets an update of its own view on the world. As mentioned in section 2.1, this view covers only a small part of the environment around the agent. A special synchronization module is responsible for the timing of the updates of the agents' views. We explain this synchronization process in detail later.

As an alternative, the agent might "know" something about the world in order to take a decision what to do. It is for example not necessary that an agent "sees" the destination for his packet if he "knows" the location of the destination. In our model we therefore endowed an agent with a ***belief base***. This belief base contains records with information that the agent has collected in the past. It is clear that some of this information is volatile. A destination for a particular color of packets will never change, but a packet located at a certain field might have disappeared after a while. In our model agents "know" which beliefs unconditionally can be trusted and which are not trustable. Agents revise suspicious beliefs as soon as they get information about them from their percept update, i.e. as soon as the subject of the belief comes inside a certain range of their vision.

2.4 A job and the states of the world

At start time the packet-world is in an initial state. Packets are scattered over the grid, and the agents are located between them. The counter that measures the agents' performance is initialized to zero. We define a ***job*** as the task of the agents to collect all the packets and deliver them at the right destinations. A job starts when a synchronization module triggers the environment to send the agents their view. Driven by their goals, agents select an action to perform. These actions result in a transformation of the state of the packet-world. This cycle repeats until the whole grid is cleared. Each time the action of an agent modifies the state of the world the performance counter is increased. As soon as all packets are delivered at their destination the synchronization module stops the process of updating the view of the agents. This informs the agents that the job has come to an end. The packet-world is then in the end state. The transformation of the world can be described as a dynamic process that transforms the initial state of the world along a sequence of discrete states into the final state by means of performing synchronized actions of the agents.

2.5 Conflict resolution and synchronization

When the agents act, the environment reacts. Thereby it takes the influences of the agents into account and produces a new consistent world. In our model we distinguish between two levels of

synchronization. First we have synchronization at the level of concurrent actions. We call this ***system synchronization.*** This level of synchronization guarantees that the "laws of the world" are respected. For example, only one agent at a time can step to a particular free field. System synchronization is implicitly integrated into a CPN. A second level of synchronization is situated at a higher level of interaction between agents. We call this the level of ***functional synchronization***. Functional synchronization offers support for coordination of actions between agents. All actions of the agents are synchronized in ***action cycles*** as shown in Figure 2.

A cycle starts with updating the perception of each agent. Based on its state and the new percept he receives, each agent then can reason about what he wants to do. The agent selects an action and produces an influence invoked on the environment. The environment calculates the reactions of all performed influences and notifies the agents by means of a consumption for each of them. As soon as all reactions are completed the environment will be triggered to calculate a new percept for each agent, and that starts a new action cycle. In our model functional synchronization is realized by means of the synchronization module.

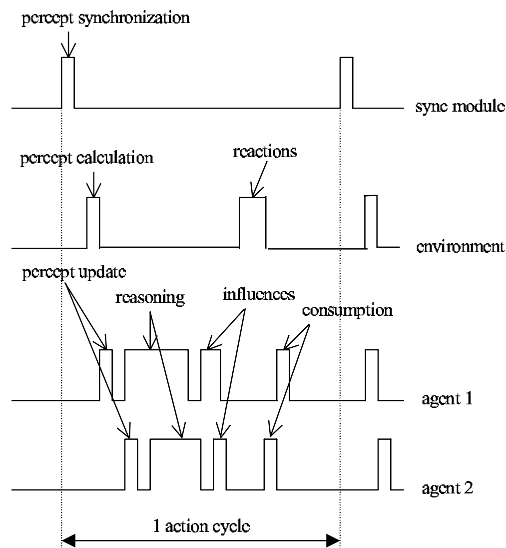


Figure 2. Functional synchronization.

One might wonder why we decided to introduce functional synchronization. After all it limits the freedom of action of the agents. Agents are no longer allowed to handle on their own rhythm. But this is just the point. The problem solving power of a multi-agent system arises from the interaction between the agents of the system. In order to cooperate, agents have to coordinate their actions. Coordinating actions between two (or more) autonomously running agents is hard to achieve. Therefore we introduced functional synchronization. For the price of some individual freedom we offer the agents a clean framework to coordinate their interactions. We fit in communication into this model. This means that sending a question or responding to it by means of sending an answer, are both integrated into the extended model as first class actions.

3. Modeling the basic components of the packet-world

In this section we discuss how we modeled the basic model of the packet-world by means of a CPN. First we give a high level overview of the model. This identifies the different modules of the multi-agent system. Next we discuss the CPN for each separated module.

We bring the separated modules together in a global net after discussing the integration of communication into the basic model in section 4.

3.1 High level model

We have divided the basic model for the packet-world into three separated modules, each representing one fundamental component of the packet-world. We distinguish between the environment (box), the agents (rounded boxes) and the synchronization module (diamond).

As shown in Figure 3 agents in the basic model only interact with the environment. The white arrows represent the influences performed by the agents. The gray arrows represent the consumptions and percepts for the agents. The synchronization module regulates the updates of the latter.

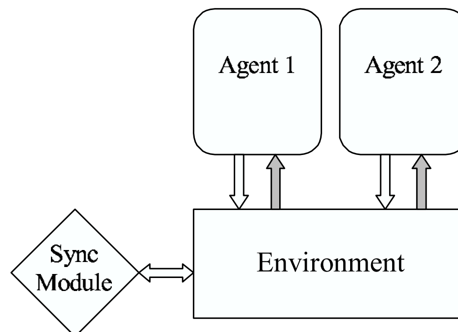


Figure 3. High level basic model with 2 agents.

Before we go into the separate CPNs for the different modules we first have to tell something about our approach for modeling modularity. In each module there are two kinds of places. There are circles that represent ***internal places*** and ovals that represent ***interface places***. Interface places are similar to the notion of ***fusion places*** as defined in [4].

Different modules can be combined with each other by merging overlapping interface places. Note that we distinguished only between internal and interface places to indicate that some places of a module will overlap with similar places of other modules when a global CPN is composed. The graphical distinction has no particular semantic meaning related to places of a CPN in general.

3.2 Model of the environment

The environment models the world in which the agents live. For our packet-world we modeled the environment as one centralized entity. The agents can interact with the environment by means of a set of actions. The concurrent actions of the agents lead to the modification of the world. Agents are notified of those transformations by means of (i) consumptions (i.e. what they get from their invoked influences) and (ii) percepts (i.e. a partial view of the state of the world around the agent). The environment keeps track of how efficient the agents perform their job. We have modeled this efficiency tracker as a simple counter that is incremented each time an agent invests a relevant portion of energy, i.e. makes a step or moves a packet. Figure 4 shows the CPN for the environment.

The data of the environment is modeled as a token of the colorset ***World***, located in the place ENVIRONMENT. This token is a list of ***Item***, each item being a record with two components:

```
color Item = record name:Name * coord:Coordinate;
color World = list Item with 1..(worldsize*worldsize);
```

The performance efficiency of the agents at a certain point in time is modeled as a set of anonymous tokens collected in the place COUNT.

The reactions of the environment are modeled as transitions. A reaction takes an influence and the state of the world as input. In case the reaction produces a successful action the involved part of the world is modified. Otherwise the world is left untouched.

Furthermore a reaction produces a consumption for the agent that is sent to the corresponding interface place and a synchronization token that is sent to the synchronization module.

Whether an action ends successfully or not depends on the actual state of the world. This is tested by means of the guards of the transitions. Let us look at one example.

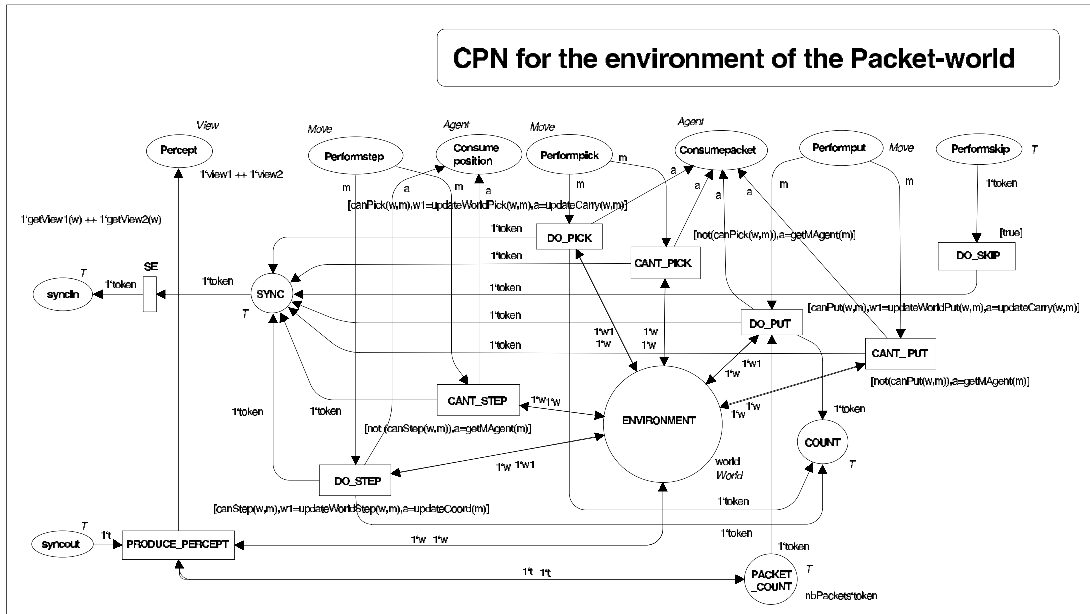


Figure 4. CPN for the environment of the packet-world.

Suppose an agent intends to make a step. Therefore he puts a token m of the colorset *Move* into the interface place “Performstep“. The token m contains information about the identity of the agent and the coordinate of the square that he wants to step to. The reaction of the world will be one of the two following possibilities:

1. If the action succeeds the transition DO_STEP with the following guard will fire:

```
(*w models the world, m the invoked influence and a the consumption *)
[canStep(w,m),w1=updateWorldStep(w,m),a=updateCoord(m)]
```

2. If the action fails the transition CANT_STEP will fire:

```
[not(canStep(w,m)),a=getMAgent(m)]
```

In the first case the condition `canStep(w,m)` is fulfilled and the state of the world as well as the position for the agent are updated. In the second case `canStep(w,m)` fails. In this case the original location of the agent is copied into the consumption for the agent.

The last part of the environment concerns the production of the agents’ percepts, modeled as the transition PRODUCE_PERCEPT. As soon as a token arrives at the "syncout" place of the synchronization module, this transition fires. It reads the world, produces the agents’ updated views and puts them in the interface place “Percept”. There the agents can pick them up. Note that the environment only produces percepts as long as the produce percept transition can read a token from the PACKET_COUNT place. Initially this place contains one anonymous token for every packet on the grid. Later on, each time an agent delivers a packet on its destination, one token is consumed from the packet count place. Finally when the latest packet is delivered there remain no longer tokens in the packet count place and that ends the production of new percepts.

3.3 Model of a basic agent

Agents are the active entities of the packet-world. Each agent is endowed with a number of operations in order to act in the environment. He can perceive information and use it instantly or

register it for later use. He can act in the environment and manipulate things. The CPN model for a basic agent is shown in Figure 5.

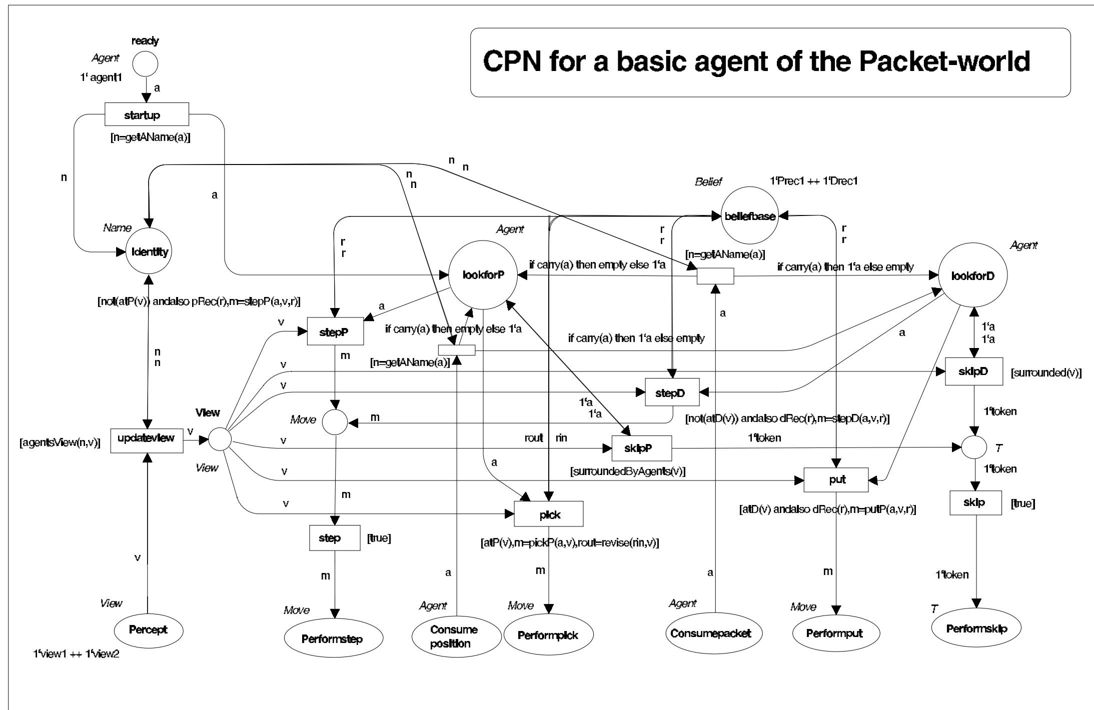


Figure 5. CPN for a basic agent of the packet-world.

All actions an agent undertakes are driven by a set of goals. The goals of our basic agent are quite limited. In case the agent hands are free he will look for a packet and pick it up. As soon as the agent holds a packet he will look for the destination and deliver it there. All actions available for an agent to fulfill its first goal (go for a packet) can only be started when a token of the colorset Agent is located in the place "lookforP". Performing one of the actions available to fulfill the second goal (deliver a packet) requires an Agent token in the "lookforD" place. An Agent token contains the state of the agent he maintains about himself. Such a token consists of three parts:

```
color Agent = record name:Name * coord:Coordinate * carry:Name;
```

Initially the Agent token is located in the place “ready”. When the execution starts, the “startup” transition fires. This passes the Agent-token to the “lookforP” place. At the same time the name of the agent is placed in the “identity” place. The agents’ identity will be used later on to dispatch the percepts of the environment to the various agents.

The state an agent maintains about the world around him is modeled as tokens of the colorset *Belief* stored in the place “beliefbase”:

```
color BeliefSubj = with pRec | dRec;
color Belief = record subj:BeliefSubj * item:Item;
```

In our basic model a belief contains information about an item of the world (for the definition of an Item, see section 3.2). We have provided two kind of beliefs, one for a packet (subj = pRec) and one for a destination (subj = dRec). Our basic agents actually use their belief base only passive and in a limited way. In fact they will only look for a packet or the destination of a packet in the base when they do not see it. So only when the programmer has given the agents some initial information (by

means of an initial marking of the belief base) they take profit of their belief base, otherwise it is of no help.

An agent trusts the beliefs about a destination but he revises beliefs about a packet as soon as he approaches the subject of the belief. This is done by means of the `revise(Belief,View)` function in the guard of the transition "pick".

The agents view on the world is modeled as tokens of the colorset *View*:

```
color View = list of Item with 1..(worldsize*worldsize);
```

In practice this list never contains all items of the world. The head of the list is always the item that corresponds to the agent himself. Thereafter the environment copies only the items around the agent in a range defined by the variable "view-size". Figure 6 illustrates the limited view an agent has on the world. In this example the size of the view is 2.

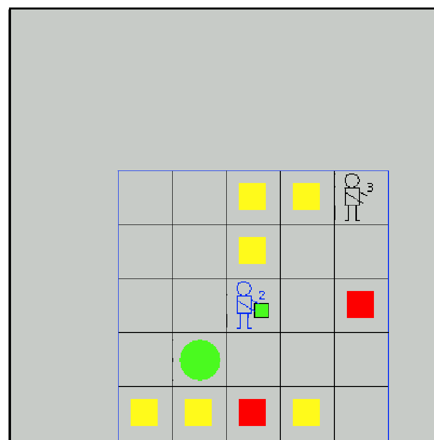


Figure 6. View on the world of Agent 2.

Now we discuss the action set of the agent. Each action is modeled as a transition. Such a transition consumes at least one token of the colorset Agent and one of the View set. Optionally the belief base is consulted. If the Agent token is located in the "lookforP" place the agent can make a step, pick up a packet or skip. If the token is in the "lookforD" place the agent can make a step, put its packet down or skip. In each case, the selection of the action is based on the criteria described in the guards of the transitions. We illustrate this for the action "put":

```
(* v models the view of the agent, a the intern data of the agent, r
one of the beliefs of the agent and m the invoked influence *)
[atD(v) andalso dRec(r),m=putP(a,v,r)]
```

The action "put" will be selected only if the agent is next to the destination of the packet he carries, i.e. the `atD(v)` condition. The record with possible information about the destination is selected from the belief base with the `dRec(r)` condition. If this record contains the coordinate of the destination, the agent creates an influence m at once, delivering its packet. If the destination is unknown he searches it from its actual view and creates a similar influence. The influence m is sent to the "Performput" interface place, where the environment takes it up for handling. An influence is modeled as a token of the colorset *Move*:

```
color Move = product Agent * Coordinate;
```

This tuple contains the Agents' identification (see section 3.3) and the coordinate (i.e. a tuple (x,y)) of the square where he intends to perform some influence. A Move token together with the place where it lands offers the environment enough information to determine the action an agent intends to

perform. The percepts from the environment come from the interface place "Percept". As soon as a new percept arrives the agent must identify himself in order to obtain the new information. Therefore the "updateview" transition reads the agents' identity. If there is a match, the view will be accepted and broadcast over the possible actions of which one is selected for execution during the next action cycle. The reactions to the influences are consumed from "Consumeposition" and "Consumepacket". Here a similar identification scenario is used. After accepting a consume the Agent token is directed to one of the main places "lookforP" or "lookforD" according to the fact the agent carries a packet or not.

3.4 Model for the synchronization module

The synchronization module models the notion of *functional synchronization* as we already described in section 2.5. It offers the agents an implicit framework for coordinating their interactions. Figure 7 shows the module with its connections to the environment.

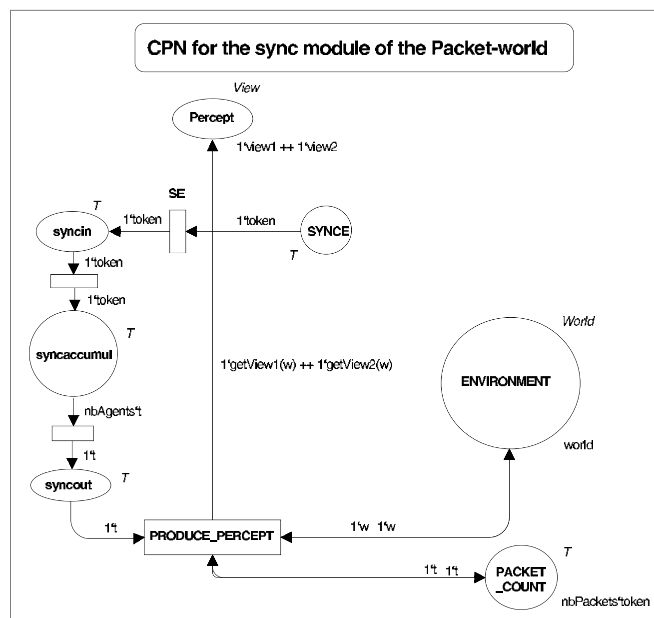


Figure 7. Synchronization module for the packet-world.

The behavior of the module is straightforward. Each time the agents performs their actions, the environment will react on them. For each reaction in particular, an anonymous synchronization token is produced and placed in the "SYNCE" place. These tokens are sent to the "syncin" interface place of the synchronization module. From there on, the tokens are collected in the "syncaccumul" place. When the actions for all agents are handled this place contains a number of tokens equal to the number of agents living in the packet-world. This triggers the output transition to fire, placing an anonymous token in the "syncout" place. On his turn this enables the PRODUCE_PERCEPT transition of the environment, such that new percepts can be calculated and sent to the agents. This starts a new *action cycle* (see section 2.5).

3.5 Complete CPN for the basic version of the packet-world

With de separated modules we now can compose the complete CPN for the packet-word. This model, depicted in Figure 8, gives a detailed picture of the high level model we presented in Figure 3. All interface places are combined according to the techniques we mentioned earlier in the paper. To keep a clear overview we limited the number of agents to two. In general however, a MAS may be composed of much more agents. In our model each agent has its own CPN module.

Figure 8. CPN for the basic model of the packet-world.

If we want to model a MAS with more agents we can combine the modules in a hierarchical CPN. However we do not discuss this further in this paper. To illustrate the purposes of this paper a model with two agents is sufficient.

4. A CPN for communicating agents

In this section we extend the basic model of the packet-world in a way the agents can communicate information with each other. Communication enables agents to coordinate their actions and behavior, resulting in a multi-agent system that is more coherent. Agents can use their abilities to communicate to better achieve the goals they are driven by.

Communication is part perception (the receiving of messages) and part action (the sending of messages). The conversation between two agents follows a protocol. A protocol enables agents to exchange and understand messages. To extend a basic agent with functionality for communication we build a "communication module" that can be plugged into the basic model of that agent. Furthermore the environment must be equipped with infrastructure to handle messages (mail). Therefore, we build a "postal service module" that can be plugged into the environment.

In this section we first introduce the communication protocol for our agents. Then we give a high level overview of our extended model for the packet-world, including communication. Next, we present the new modules necessary for communication and conclude with the complete CPN for the extended model.

4.1 Communication protocol

Basically the agents in our packet-world all have the same capabilities. This is reflected in the roles they play in a dialogue. As long as an agent "sees" another agent he is capable of sending a message to that peer colleague. For the moment we only model question/answer types of messages. In particular we limit the subject of the messages to requests for information. Figure 9 shows the different steps in a dialogue. The syntax of the protocol is described in section 4.3.

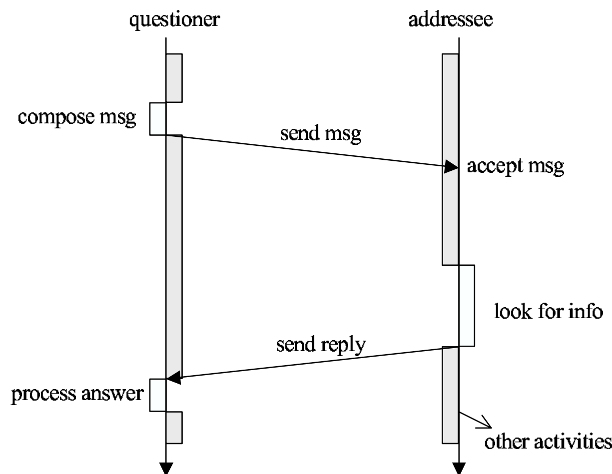


Figure 9. Communication among the agents.

An agent delivers a composed message at the "inbox" of the postal service. This service has knowledge of the mailboxes of the agents and routes the message to the mailbox of the addressee. As soon as the message arrives, the addressee can pick it up from his mailbox. In our model an agent is not obliged to handle an incoming message at once.

When the addressee decides to read the message, he will look for the requested information. If he knows the information he sends an answer, otherwise he informs the requester he can't help him for

the moment. When the reply arrives the information will be processed and possibly update the belief base of the requester.

4.2 High level model of the packet-world with communication extension

Figure 10 shows a high level model for the packet-world in which agents are equipped with functionality to communicate. The basic agents of our basic model are now extended with a communication module. This module permits an agent to interact with a colleague.

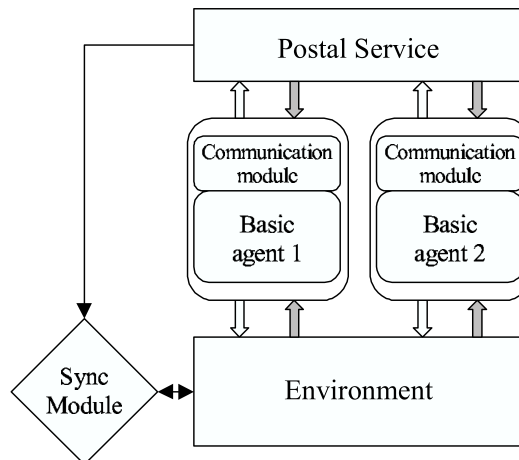


Figure 10. High level model of with functionality for inter-agent communication.

The arrows above the agents model the communication channels with the postal service module. This latter is responsible for delivering posted messages in the mailboxes of the addressees. Note that the postal service module too can produce functional synchronization pulses.

4.3 Model of the communication module for the agents

The communication module assembles functionality for an agent to send requests, respond to questions and process answers. The CPN for such a module is depicted in Figure 11.

In our model, agents can gather information from a colleague about the location of a packet or a particular destination according to its actual state. Asking for information is modeled as a transition, respectively "askforP" and "askforD". To fire one of these transitions (i.e. compose a message) a number of conditions must be fulfilled.

These conditions are described in the guards of the transitions. Let us look to one example, the "askforD" transition:

```
(* a models the internal state of the agent, v its current view *)
[canCallD(a,v),question=askForD(a,v)]
```

The function `canCallD(a,v)` returns true only if (i) the agent actually does not see the destination of its packet and (ii) he sees a colleague on which he can ask the information. If `canCallD(a,v)` succeeds, the function `askForD(a,v)` produces a question that as a token of the colorset Message is delivered in the inbox of the communication module.

A Message has the following structure:

```
color Performative =
  with questP | answP | noanswP | questD | answD | noanswerD;
color Message =
  record from:Name * to:Name * perform:Performative * content: Item;
```

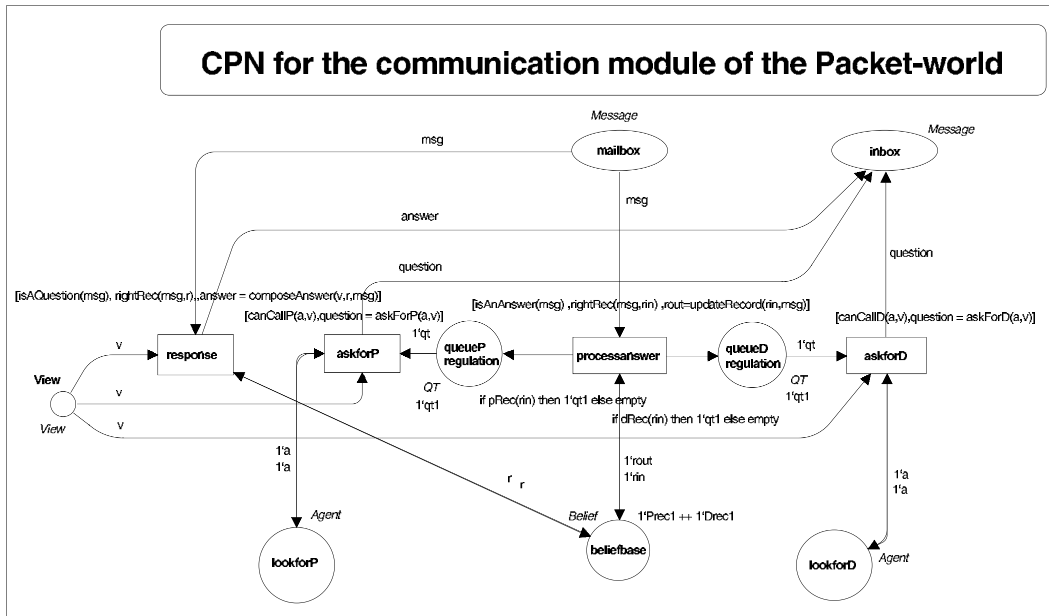


Figure 11. Communication module for an agent.

The performative informs the addressee about the type of message that is been sent. The content of a question is an item structure that has to be completed by the addressee. If for example an agent "a1" asks an agent "a2" for the location of the destination for yellow packets he will compose the following message:

```
{from="a1",to="a2",perform=questD,item={name="yellowDest",coord=null}};
```

When agent "a2" receives this message he knows exactly what "a1" is asking for. He will uses his belief base and actual view to find the coordinate of the "yellowDest". If he finds e.g. at coordinate (4,3) the yellow destination, he replies the following message :

```
{from="a2",to="a1",perform=answD,item={name="yellowDest",coord=(4,3)"}}
```

Replying to a message is performed in the transition "response". To fire this transition a view is consumed together with the message from the mailbox. The information in the believe base is only consulted as an extra information source.

The model prevents an agent to send messages for information over and over again. The "queueregulation" places contain a limited number of tokens that are consumed each time a question is sent and only restored when the answer is processed. Processing an answer comes down to update the belief base for the case the answer contains new information, otherwise the answer is thrown away.

4.4 Model of the posting service module

The postal service is responsible for delivering the mail of the agents at the right mailbox. Figure 12 shows the CPN for this module. The postal service has one "inbox" place where agents can leave their messages. Each message is accepted in the transition "acceptmsg". This transition puts the message in the "msgbuffer" place and produces three other tokens. The first token goes to the "msgcount" place where the user can read the total number of messages the postal service has handled so far. The second token goes to the "msglog" place, where a log of all messages is saved. The third token is sent to the "syncP" place from where it is directed to the "syncin" place of the synchronization module. This means that in our model, sending messages is coordinated with the

other actions agents can perform. This fits in our concept of functional synchronization, offering a solid base for the agents to coordinate their activities.

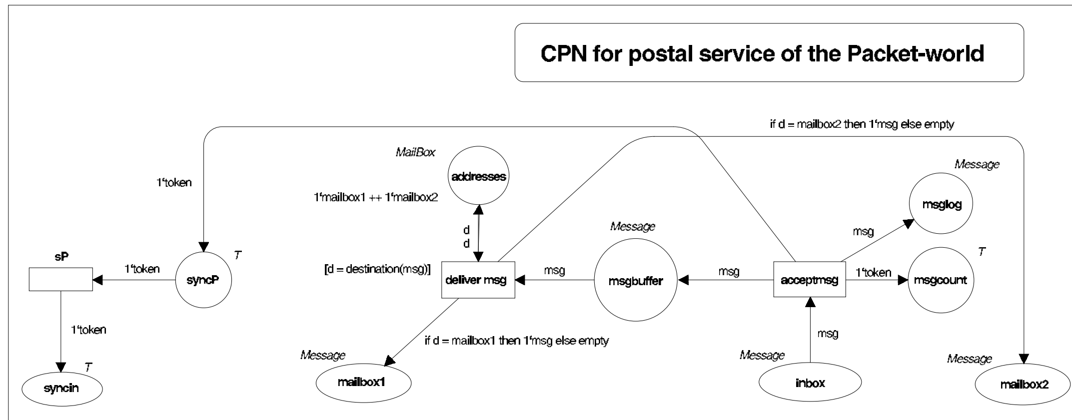


Figure 12. Postal service module.

When a message resides in the “msgbuffer” it is delivered to the addressee by firing the “delivermsg” transition. This transition consults the “addresses” place where the references to the different mailboxes are stored. Based on the mapping between the addressee indicated in the message and the information of the mailbox references, the message is delivered in the mailbox of the addressee where he can pick it up later on.

4.5 Complete CPN for the packet-world with communicating agents

With de separated modules we have proposed in the previous sections, we now can compose the complete CPN for the packet-word with communicating agents. This model, depicted in Figure 13, gives a detailed picture of the high level model we presented in Figure 10.

5. First experiments

In this section we briefly give an overview of the results of our first experiments with the CPNs for the basic version of the packet-world and the extended version with functionality for communication. We first discuss results of simulations; next we look to a number of verifications.

5.1 Simulations

With the Design/CPN tool a CPN can be executed, automatically or interactive. This allows us to follow the successive actions of the agents. We did tests on a world with size 5 and one with size 8. For both we changed the view-size for the agents. Table 1 gives an overview of the results. The numbers are rounded averages for 5 jobs.

Table 1. Simulation results.

World	view-size	Kind of model	COUNT	% gain	msgcount
world-size = 5 nbAgents = 2 nbPackets = 5	2	basic	26	31	--
		communication	18		4
	3	basic	15	7	--
		communication	14		2
world-size = 8 nbAgents = 2 nbPackets = 16	3	basic	167	23	--
		communication	129		16
	4	basic	110	2	--
		communication	108		2

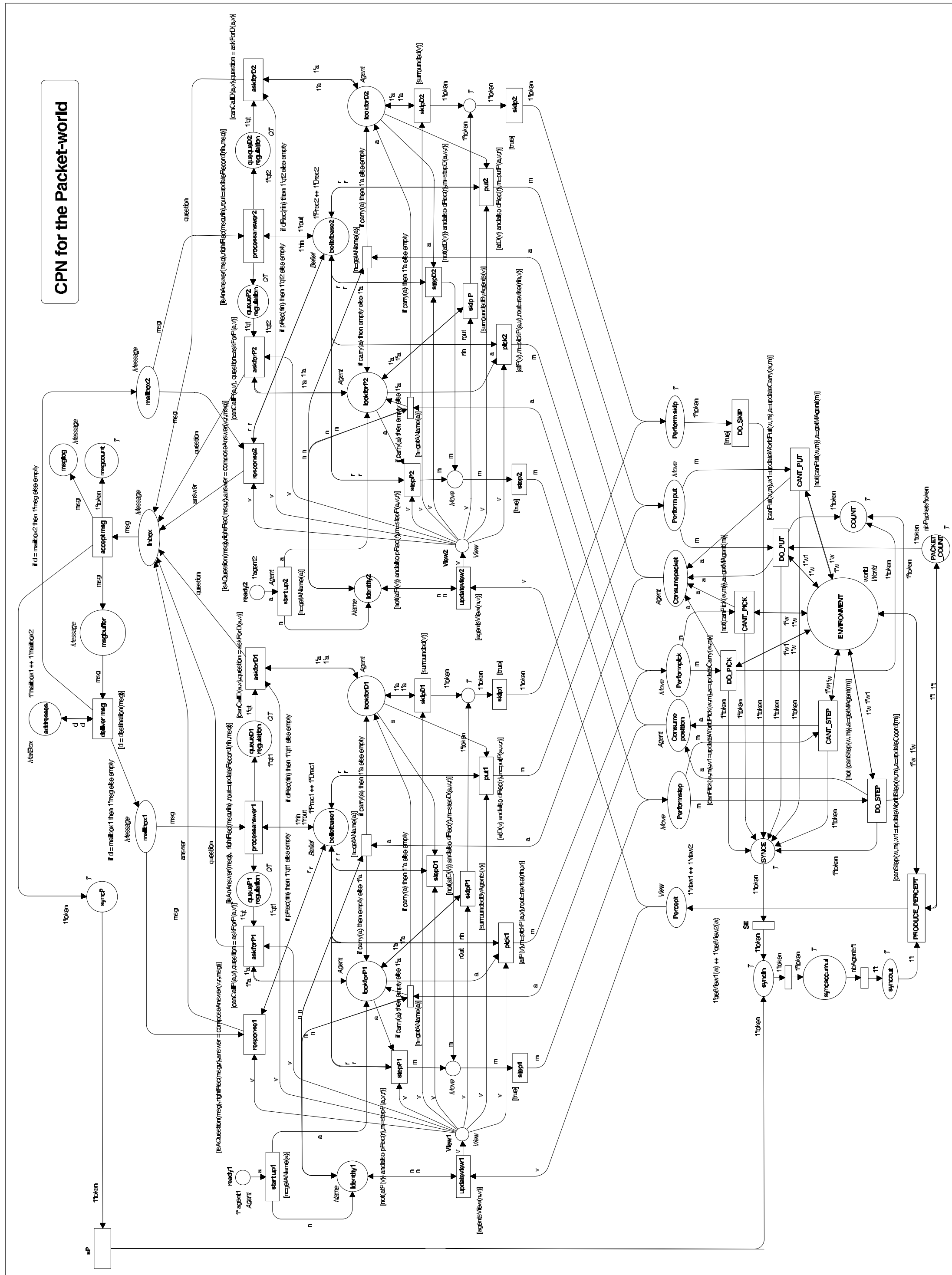


Figure 13. Complete CPN for the packet-world with two communicating agents.

If we compare the results for one kind of model (basic or communication), we see that increasing the view-size significantly reduces the number of steps the agents need to complete a job. The obvious explanation is that a greater view-size increases the information for the agents, so they can act more efficiently. When we compare the results between the two kinds of models for one view-size we notice that for small view-sizes the communicating model scores significantly better then the basic version. For greater view-sizes the gain is only marginal. In the first case agents communicate information to each other, so they can act more efficiently. In the latter case the view of the agents covers a great part of the world, so they mostly "see" what they are looking for, and there is no need to request information from each other. Our first tests confirm the value of information interchange between agents, but for better-founded conclusions we need to do more tests, especially with greater worlds and more agents.

5.2 Verifications

Besides simulation, the Design/CPN tool offers support for formal verifications of CPNs by means of the Occurrence Graph Tool [4]. An occurrence graph is a directed graph with a node for each reachable marking and an arc for each occurring binding element. With such occurrence graphs we did a number of formal verifications for the CPNs of the packet-world. We discuss here some results for the basic version of a world of size 5 with 2 agents that have to collect 5 packets.

The tool generates a standard report (for more information see [5]) that already gives a lot of information. E.g., the "Liveness Properties" gives the "Dead Transitions Instances" for the occurrence graph. If e.g. CANT_PICK is such a dead transition instance this means that for the given packet-world there where no conflicts between the agents with picking up packets. Contrary if e.g. CANT_STEP is not a dead transition instance we are sure both agents must have stepped at least once to the same square. To investigate the CPN in more detail the occurrence graph tool offers a lot of standard query functions. Besides, users can formulate their own customized queries too. To do a number of formal verifications, we extended the CPN for the packet-world with an extra "test-module", depicted in Figure 14.

We added four more places to the Petri Net, PACKETS_ON_GRID, CARRIED_PACKETS, DELIVERED_PACKETS and FINISCH_JOB, as well as two more transitions FINISH_JOB and TEST_END_JOB. Initially PACKETS_ON_GRID contains *nbPackets* anonymous tokens, while CARRIED_PACKETS and DELIVERED_PACKETS are empty.

When an agent picks up a packet (i.e. DO_PICK fires) one token from PACKETS_ON_GRID is passed to CARRIED_PACKETS. When an agent delivers a packet at its destination (i.e. DO_PUT fires) the token is further passed from CARRIED_PACKETS to DELIVERED_PACKETS. At the end of the job all packets are delivered, so PACKETS_ON_GRID and CARRIED_PACKETS are empty, while DELIVERED_PACKETS contains *nbPackets* anonymous tokens. When for each agent (during the final action cycle) the Agent token reaches the "lookforP" place and a new synchronization token reaches the "syncout" place, the FINISH_JOB transition is enabled and will fire. This clears the Petri Net and an anonymous token arrives in the END_JOB place. This enables the TEST_END_JOB transition that from then on will fire forever.

The packet-world is free of deadlocks. To prove that no deadlock appears we have to prove that there exists a path from each node in the occurrence to the node that represents the final marking, representing the state in the END_JOB place. That particular node, the leaf node of the occurrence graph, is shown with its predecessors in Figure 14. The proof is straightforward. The SearchNodes function "PROOF 1" in Figure 14 searches the number of nodes that have no path to the leaf node. Since this number is zero we have proven that the packet-world is deadlock free.

Figure 14. Test module for the packet-world.

A job is correctly solved in a limited number of steps. To prove that a job of the packet-world is correctly solved in a limited number of steps we have to take two steps. First we have to prove that the following place-invariant holds:

> *"the sum of anonymous tokens (each representing a packet) for the places PACKETS_ON_GRID, CARRIED_PACKERTS and DELIVERED_PACKETS is constant and equal to nbPackets in each node of the occurrence graph, except in the leaf node"*

This invariant tells us that neither strange packets enter the packet-world, nor any packet is lost during a job. The SearchNode function of "PROOF 2" in Figure 14 shows that the invariant holds. To complete the proof we must demonstrate that the number of steps to reach a solution (i.e. the TEST_END_JOB is enabled) is limited.

Since "PROOF 1" tells us that there exists a path from each node in the graph to the leaf node, we can conclude that execution always ends in a limited number of steps.

6. Conclusions and future work

In this paper we presented a CPN for the packet-world, a multi-agent application. In our research we use this application as a case to study the fundamentals of sociality in MASs.

Let us now reflect and verify that our expectations from using CPNs have been worked out. An important argument for using CPNs was its strong graphical expressiveness. We build up the packet-world by means of compositional modules. When we integrate communication infrastructure into the basic model we got a clear view on how this impacts the agents and the environment. Building an executable CPN leaves *no escape* for the designer. Every aspect must be modeled explicitly and unambiguously. Therefore we are forced to find concrete solutions for several problems. One typical example is the way we realized functional synchronization. One can talk al lot about such an aspect, but modeling it in a CPN brings the designer to the very *essence* of it. As such, we can state that we learned a lot about MASs, using CPNs to model them. Another argument why we have chosen CPNs was the possibility of simulation. Simulating a MAS like the packet-world can be done in different ways. Executing a CPN is not always the most attractive way to

simulate such a problem. But in fact, that is not the point. What is important is the fact that the execution of a CPN is a direct simulation of the ***model itself***. So the simulation directly shows us the value of the model we have built. A last argument for CPNs we mention here is the possibility of formal verification. The MAS community has a strong tradition in formal description and verification of its ideas. CPNs join this approach. Formal verification lets the designer proof the correctness of (parts of) his model. Without the Design/CPN tool it would be very hard to prove that our packet-world has a correct solution in a limited number of steps. With the tool it is quite simple to proof this property.

This paper reflect our first experiences with CPNs as a tool to model agents' sociality. The model we have developed forms a solid basis for future research of agents' social behavior. We conclude with some thoughts about our future work. It is our intention to build modules for a number of other social skills for the agents of the packet-world. Examples are agents that cooperate by forming a chain and passing packets to each other, or agents that coordinate their actions avoiding future conflicts (e.g. 2 agents who both step a long way to the same packet). Building such models will gain us more in-depth knowledge about the fundamentals of sociality in MASs. To manage the complexity of extensive models we can use hierarchical CPNs. Later on we intend to generalize the insights we learned from the packet-world. We intend to build abstract models for different classes of social skills. The aggregate of these models can serve as a well defined and easy to communicate formal model for social agents in MASs.

7. References

[1] C.A.Petri, "Communication with Automata", Vol.1. Applied Data Research, Princeton, AF 30(602)-3324, 1966.

[2] K. Jensen, "Coloured Petri Nets", Lecture Notes Comp. Science, nr. 254, Advances in Petri Nets, Bad Honnef, 1986.

[3] J. Ferber. "Multi-Agent Systems, an introduction to distributed artificial intelligence", Addison-Wesley, ISBN 0-201-36048-9, 1999.

[4] Design/CPN, A Computer Tool for Coloured Petri Nets http://www.daimi.aau.dk/designCPN/ .

[5] K. Jensen, Coloured Petri Nets. Basic Concepts, Springer Verlag, 1992, ISBN: 3-540-60943-1.

[6] N. Jennings, On agent-based software engineering, Artificial Intelligence, 117 (2) 277-296, 2000.

[7] Huhns, Stephens, Multi-agent Systems and Societies of Agents, in G. Weiss, Multiagent Systems, MIT press, 1999.

[8] Y. Shoham, M. Tennenholtz, On Social Lows for Artificial Agent Societies: Off-Line Design. Agents 1998.

[9] R. Conte, C. Castelfranchi, Simulations understanding of norm functionality's in social groups, 1993.

[10] Goldman, Rosenschein, Emergent Coordination through the Use of Cooperative State-Changing Rules, DAI, 1994.

[11] J. Kittock, Emergent Conventions and Structure of MAS, Complex Systems Summer School, Santa Fe 1995.

[12] A. Walker, M. Wooldridge, Understanding the Emergence of Conventions in Muli-Agent Systems, ICMAS'95.

[13] P.R. Cohen, H.J. Levesque, Teamwork, Special Issue on Cognitive Science and Artificial Intelligence, 1991.

[14] P. Panzarasa, N.R. Jennings, T.J. Norman, Social Mental Shaping: Modeling the Impact of Sociality on the Mental States of Autonomous Agents, Computational Intelligence 17 (4) 738-782, 2000.

[15] P. Panzarasa, N. Jennings, The organization of sociality: a manifesto for a new science of multi-agent systems, Proc. 10th European Workshop on Multi-Agent Systems (MAAMAW-01), Annecy, France, 2001.

[16] T. Holvoet, An approach for open concurrent software development, PhD thesis, K.U.Leuven, Belgium 12/1997.

[17] J. Fernandes, O. Belo, Modeling Multi-Agent Systems through Colored Petri Nets, 16th IASTED International Conference on Applied Informatics (AI'98), Garmisch-Partenkirchen, Germany, pp. 17-20, Feb/1998.

[18] M. Costa Miranda, A. Perkusich, Modeling and Analyses of a Multi-Agent System using Colored Petri Nets, In Workshop on Applications of Petri Nets to Intelligent System Development, Williamsburg, USA, June 1999.

[19] D. Moldt, F. Wienberg, Multi-Agent Systems based on Coloured Petri Nets, Azéma und Balbo 1997, 1997.

[20] R. Cost et al., Modeling Agent Conversations with Colored Petri Nets, IJCAI '99, Stockholm, Sweden, 1999.

[21] M. Duvigneau, D. Moldt, H. Rolke, Concurrent Architecture for a Multi-Agent Platform, in Proceedings of AOSE'02, AAMAS, Bologna Italy, July 2002.

[22] Miyamoto et al. wrote a number of articles in Petri Nets Newsletter, http://www.informatik.uni-hamburg.de/TGI/pnbib/keywords/a/agent_net.html

Modelling Mobility and Mobile Agents using Nets within Nets

Michael Köhler and Heiko Rölke

Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Str. 30, D-22527 Hamburg
roelke@informatik.uni-hamburg.de

Abstract. Mobility creates a new challenge for dynamic systems in all stages of modelling, execution, and verification.
In this work we present an application area of the paradigm of "nets within nets". Nets within nets are well suited to express the dynamics of open, mobile systems. The advantages of Petri nets – intuitive graphical representation and formal semantics – are retained and supplemented with a uniform way to model mobility and mobile (agent) systems.
First the modelling of mobility is introduced in general, the results are carried forward to model mobility in the area of agent systems. The practicality of the approach is shown in a second step by modelling a small case study implementing a household robot system.

Keywords: agent, mobile agent system, mobility, nets within nets, Petri nets, Mulan, Renew

1 Introduction

The general context of this paper is mobility in open multi agent systems. The main question is how to model mobility in an elegant and intuitive manner without losing formal accuracy. The modelling language should feature a graphical representation to be used in a software engineering process. The modelling paradigm should be capable of expressing the different kinds of agent mobility. The models should build upon a formalism that has a formal semantics to support verification and execution. The (direct) execution of the models prohibits errors of manual translation from models e.g. to program code. Executable models support the validation process.

Here, we present a proposal, how the paradigm of "nets within nets" can be used to describe mobility. The paper consists of two parts: First it is shown how "mobility" in general can be expressed (environment, moved entity, different types of movement). In the second part these results are used to support the engineering of mobile agent systems.

Most of the work on mobility is based on calculi. Mobility calculi like in [CGG99], [VC98] or in [MPW92] formalise mobility without having a visual representation, thus being unsuited for the software-technical modelling of mobile entities. Mobile Petri nets [AB96] provide a description of mobility by embedding

the π-calculus into the formalism of ordinary Petri nets. Since the formalism is far from the general intuition of a token game, it cannot be used here for the proposed goal of supporting the specification process of complex software systems.

Structure of the paper In section 2 general statements on mobility are given: minimal preconditions for a formalism to express mobility are suggested as well as our needs for an intuitive modelling. A selection of formalisms is compared on this basis. We present a short introduction to the paradigm of nets within nets and show how this paradigm can be used to model mobility. In section 3 we give a description how this paradigm is used in the specific context of open multi agent systems. Section 4 presents a small case study of a mobile agent system using the modelling proposals. The paper closes with a conclusion and an outlook to further work.

2 Nets within Nets and Mobility

Figure 1 shows the mandatory elements of a system to become a *mobile* system. The overall system is divided into separate locations. At least two different locations are necessary. Locations host entities (some of) which are able (under certain circumstances) to undertake a movement from one location to another, which means that the environment of the entity changes.[1]

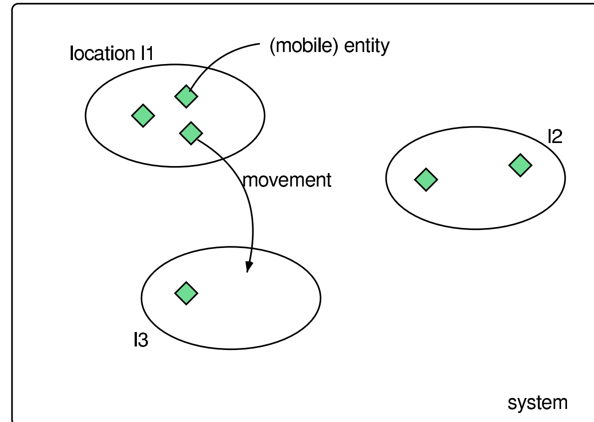


Fig. 1. Elements of mobile systems

[1] This changing may either be logical or physical. The modelling of real-world scenarios makes it necessary to cope with the additional problems of physical movements – which may be mapped to a logical changing of the environment.

An important point of mobility is the embeddedness of the mobile entity: each entity is embedded in a local environment (location) that assists the entity by offering some services and restricts it by declining others (or not having the potential to offer them). If all locations look the same to the mobile entity and no difference is made in local versus remote communication, movements are transparent to the mobile entity. This is an important feature for example in load balancing systems. The systems we are considering usually show several differences between the locations and hence are more interesting to model, since these differences cause the complexity of the systems.

Modelling opportunities To argue why we are not using a common mobility calculus, we take the Seal Calculus [VC98] as an example. In the Seal Calculus a process term P with locality is denoted as $E[P]$. Mobility is described by sending seals over channels. The term $\bar{x}^*\{y\}.P$ describes a process, which can receive a seal y over the channel $\bar{x}^*$ and then behaves like P. For receiving the term $x^*\{z\}.Q$ describes a process, which receives the seal z, substitutes this seal for every occurrence of z in Q and then behaves like this new Q.[2]

Mobility calculi like the Seal Calculus offer about the same modelling power while being (partly) easier to analyse than our extended Petri net formalism. We favour nets within nets for the description of mobility, since Petri net models are easier to read without losing the exactness. This is the main difference to models based upon calculi or logic, like HiMAT of Cremonini et al. [COZ99]. Nets within nets can be used for modelling purposes in our MULAN[3] architecture as well as for analysis purposes. Additionally, our formalisation allows the description of location and concurrency in an integrated way, which would not have been possible with simple Petri nets or mobility calculi alone.

Buchs and Hulaas [HB01] use an extended variant of high-level algebraic Petri nets to model mobile (agent) systems. The difference to this approach is the approach to mobility: Their system uses a π-calculus like style, e.g. mobility via passing of names. In our opinion it is more intuitive to remove the mobile entity from one location and let it enter a new one. This will be shown in subsection 2.2. Nevertheless, it is possible to simulate our approach with π-calculus like mobility and vice versa.

The use of widespread (graphical) modelling formalisms like UML is not possible, because all included types of diagrams (e.g. class diagrams, activity diagrams, state charts) are static and/or do not offer a notion of (dynamic) embeddedness. Additionally they lack a formal semantics. This is especially the case for the deployment diagram, that was originally used to model the *static* allocation of objects among different computers (clients and servers).

In the majority of cases mobile agents are directly implemented using a programming language like Java [Sun]. As an example we cite Uhrmacher et

[2] The Seal Calculus thus describes a *spontaneous move*. See Figure 4 in subsection 2.3.

[3] MULAN stands for MULti Agent Nets and is an approach to cover a complete agent framework (specification, design, implementation, and execution) on Reference nets and Java [Röl02].

al. [UTT00] (representative for other publications): The implementation is done using Java whilst the mobility aspect of the system is presented using simple graphics (without semantics). In contrast to this approach we try to unify illustration and implementation using the same (Petri net) model for both purposes.

Summing up these arguments we decided to use an extended Petri net formalism, that will now be introduced.

2.1 Nets within Nets

The paradigm of "nets within nets" due to Valk [Val98,Val00] is based on the former work on task-flow nets [Val87]. The paradigm formalises the aspect that tokens of a Petri net can also be nets. Taking this as a view point it is possible to model hierarchical structures in an elegant way.

We will now give a short introduction to the paradigm of Reference nets [Kum02,KW98], an implementation of certain aspects of nets within nets. It is assumed throughout this text that the reader is familiar with Petri nets in general as well as coloured Petri nets. Reisig [Rei85] gives a general introduction, Jensen [Jen92] describes coloured Petri nets.

A net is assembled from places and transitions. Places represent resources that can be available or not, or conditions that may be fulfilled. Places are depicted in diagrams as circles or ellipses. Transitions are the active part of a net. Transitions are denoted as rectangles or squares. A transition that fires (or occurs) removes resources or conditions (for short: tokens) from places and inserts them into other places. This is determined by arcs that are directed from places to transitions and from transitions to places.

Reference nets [Kum98] are so-called high-level Petri nets, a graphical notation that is especially well suited for the description and execution of complex, concurrent processes. As for other net formalisms there exist tools for the simulation of reference nets [KWD01]. Reference nets offer some extensions related to "ordinary" coloured Petri nets: net instances, nets as token objects, communication via synchronous channels, and different arc types. Beside this they are quite similar to coloured Petri nets as defined by Jensen. The differences are now shortly introduced.

Net instances Net instances are similar to objects of an object oriented programming language. They are instantiated copies of a template net like objects are instances of a class. Different instances of the same net can take different states at the same time and are independent from each other in all respects.

Nets as tokens Reference nets implement the "nets within nets" paradigm of Valk [Val98]. This paper follows his nomenclature and denominates the surrounding net *system net* and the token net *object net*. Certainly hierarchies of net within net relationships are permitted, so the denominators depend on the beholders viewpoint.

Synchronous channels A synchronous channel [CH94] permits a fusion of transitions (two at a time) for the duration of one occurrence. In reference nets (see [Kum98]) a channel is identified by its name and its arguments. Channels are directed, i.e. exactly one of the two fused transitions indicates the net instance in which the counterpart of the channel is located. The other transition can correspondingly be addressed from any net instance. The flow of information via a synchronous channel can take place bi-directional and is also possible within one net instance. It is possible to synchronise more than two transitions at a time by inscribing one transition with several synchronous channels.

Arc types In addition to the usual arc types reference nets offer *reservation arcs* and *test arcs*. Reservation arcs carry an arrow tip at both endings and reserve a token solely for one occurrence of a transition. They are a short hand notation for two opposite arcs with the same inscription connecting a place and a transition. Test arcs do not draw-off a token from a place allowing a token to be tested multiple times simultaneously, even by more than one transition (test on existence).

2.2 Modelling Mobility

The intuition of nets within nets is, that the token nets are "lying" as tokens in places just as one is used to have ordinary tokens. This is illustrated in the figures 2 and 3. When modelling more widespread nets a displaying as in the mentioned figures is not practical. Therefore the modelling tool Renew implements a kind of pointer concept: net tokens are references (hence the name) to nets each displayed in a window of their own.[4]

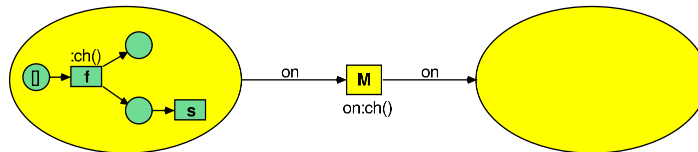


Fig. 2. Object net embedded in system net

To give an example we consider the situation where we have a two-level hierarchy. The net token is then called the "object net", the surrounding net is called the "system net". Figure 2 illustrates this situation: The object net in the left place of the system net can be bound to the arc inscription on. Doing so, transition M is activated with this possible binding. In addition M is inscribed

[4] There is another difference between the Reference nets of Renew and the intuitive modelling, namely the use of reference versus value semantics. This topic is covered by Valk [Val00].

with a synchronous channel (on:ch()). This inscription means that for the object net on to become an actual binding of transition M an adequate counterpart has to be found within on – an enabled transition inscribed with the channel :ch(). This precondition is fulfilled by transition f of the object net. So the synchronous firing of object and system net can take place and leads to the situation in Figure 3.

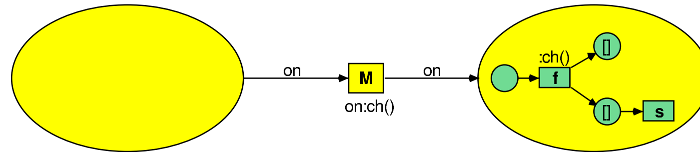


Fig. 3. Object and system net after firing

The object net is moved to the right place of the system net. Synchronously the marking of the object net changed so that another firing of transition f is not possible.

The example gives an idea how the interplay between object net and system net can be used to model mobile entities manoeuvring through a system net, where the system net offers or denies possibilities to move around while the mobile object net moves at the right time by activating a respective transition that is inscribed with the counterpart of the channel of the move transition of the system net.

Without the viewpoint of nets as tokens, the modeller would have to encode the agent somehow. This has the disadvantage, that the inner actions cannot be modelled directly, so, they have to be lifted up 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 losing the needed abstraction.

2.3 Types of Mobility

The interplay between object net and system net induces four possibilities for an object net to move or to be moved, respectively.

1. The object net is moved inside the system net, neither object nor system net controls the move. (Spontaneous Move)
2. The object net triggers the movement, the system net has no influence. (Subjective Move)
3. The system net forces the object net to move. (Transportation, Objective Move)
4. Both nets come to an agreement on the movement. (Consensual Move)

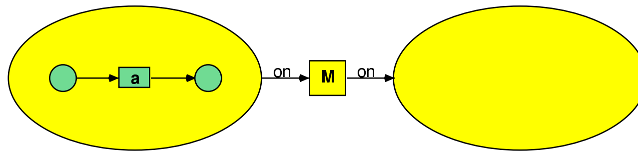


Fig. 4. Spontaneous Move

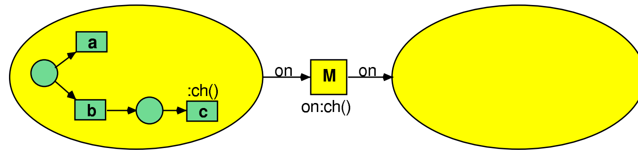


Fig. 5. Subjective Move: Object net triggers movement

If neither object net nor system net influence the movement it may happen *spontaneously*. This is the situation in Figure 4. There is no pre- or side condition for the movement.

One may argue that the second possibility does not exist, because the system net offers the ways for object nets to move from one location to another. So the system net "decides" which movements can be carried out and which can not. But in a *given* system net it is possible for an object net to control the movement as shown in Figure 5.

In the figure the only condition for the movement to be carried out is the synchronous channel (see transition M). The synchronous channel is activated if its counterpart inside the object net is also activated (that means, it is inscribed to an enabled transition). So the movement depends on the (firings of the) object net only. This movement is called *subjective* because the mobile entity itself is subject of the execution.

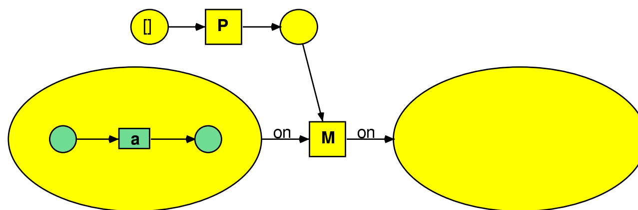


Fig. 6. Transportation: System net triggers movement

If Figure 4 is extended with some kind of condition for transition M, the system net may control the movement, the object net is *transported* from one location to another. Figure 6 shows only one possibility of how the system net may control the transition M, another one is for example a guard (see Figure 7).

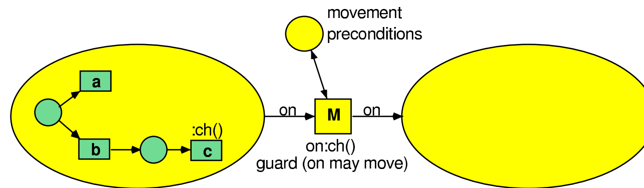


Fig. 7. Consensual Move

By combining Figure 5 and Figure 6 both object and system net influence transition M. For the transition to be enabled, they have to agree upon the movement. For this reason this kind of movement is called *consensual*. An example for such a move is shown in Figure 7.

The figure shows another way of modelling a (side) condition for transition M: a combination of a place holding movement conditions and an appropriate guard. The guard monitors the movement conditions, transition M is only enabled if the object net is allowed to move.

3 Agent Systems

In the following section we lift the general insights of how mobility can be modelled to a special form of multi agent systems. The modelling of a mobile agent (object net) moving through a "world" of several locations (system net) allows for an intuitive reproduction of real-world scenarios.

First the multi agent architecture Mulan is introduced, that also profits from the use of nets within nets.

3.1 Multi Agent Architecture Mulan

The multi agent system architecture Mulan [KMR01] is based on the "nets within nets" paradigm, which is used to describe the natural hierarchies in an agent system. Mulan is implemented in Renew [KW98], the IDE (Integrated Development Environment) and simulator for reference nets. Mulan has the general structure as depicted in Figure 8: Each box describes one level of abstraction in terms of a system net. Each system net contains object nets, which structure is made visible by the ZOOM lines.[5] The figure shows a simplified

[5] This zooming into net tokens should not to be confused with place refining.

version of MULAN, since for example several inscriptions and all synchronous channels are omitted. Nevertheless this is an executable model.

The net in the upper left of Figure 8 describes an agent system, which places contain agent platforms as tokens. The transitions describe communication or mobility channels, which build up the infrastructure. The multi agent system net shown in the figure is just an illustrating example, the number of places and transitions or the interconnections have no further meaning.

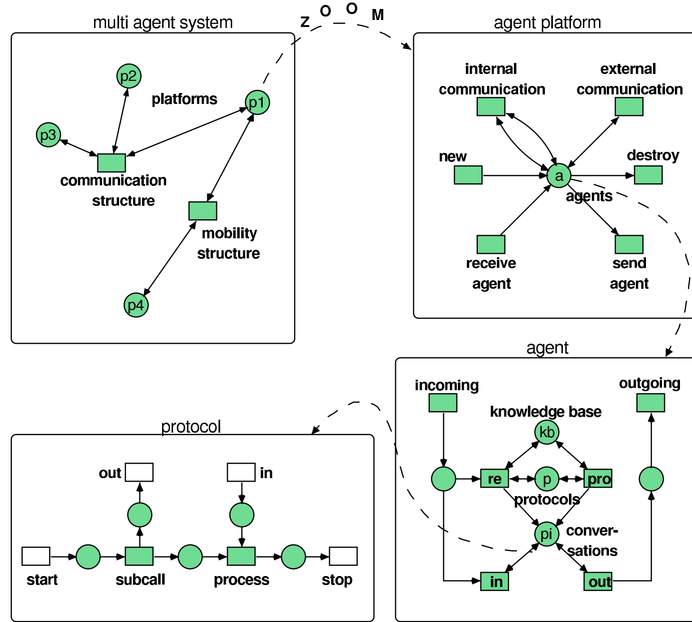


Fig. 8. Agent systems as nets within nets

By zooming into the platform token on place p1, the structure of a platform becomes visible, shown in the upper right box. The central place agents hosts all agents, which are currently on this platform. Each platform offers services to the agents, some of which are indicated in the figure.[6] Agents can be created (transition new) or destroyed (transition destroy). Agents can communicate by message exchange. Two agents of the same platform can communicate by the transition internal communication, which binds two agents, the sender and the receiver, to pass one message over a synchronous channel.[7] External communi-

[6] Note that only mandatory services are mentioned here. A typical platform will offer more and specialised services, for example implemented by special service agents.

[7] This is just a technical point, since via synchronous channels provided by RENEW asynchronous message exchange is implemented.

cation (external communication) only binds one agent, since the other agent is bound on a second platform somewhere else in the agent system. Also mobility facilities are provided on a platform: agents can leave the platform via the transition send agent or enter the platform via the transition receive agent.

A platform is therefore quite similar to a location in the general mobility scenario as it was introduced in the beginning of section 2.

Agents are also modelled in terms of nets. They are encapsulated, since the only way of interaction is by message passing. Agents can be intelligent, since they have access to a knowledge base. The behaviour of the agent is described in terms of protocols, which are again nets. Protocols are located as templates on the place protocols. Protocol templates can be instantiated, which happens for example if a message arrives. An instantiated protocol is part of a conversation and lies in the place conversations.

The detailed structure of protocols and their interaction have been addressed before in [KMR01], so we skip the details here and proceed with the modelling of agent migration.

3.2 Mobility as a system view

When modelling a complex system it is often undesirable to see the overall complexity at every stage of modelling and execution (simulation in our case). Therefore the notion of a system view is introduced. Several views on an agent system are possible, for example the history of message transfer (message process), the ongoing conversations and/or active protocols (actual dynamic state) or the distribution of agents among a system of several locations (platforms).

Using nets within nets as a modelling paradigm allows for the direct use of system models at execution time. This can be exploited as follows: The overall system is designed as a system net with places defining locations and transitions representing possible moves from one location to another. This is a direct transformation of the general mobility modelling ideas of section 2. The adaption to host platforms (that encapsulate the mobile agents) instead of agents directly is straightforward (adding one level of indirection) and can be omitted here.

Figure 9 shows an example of such a system net. Places are locations (rooms) in or in front of a house, transitions model possible movements between the rooms. The walls of the house are drawed in for illustrationary purposes only. This example is carried forward in the next section (4).

The interesting point is that it is possible to "decorate" the places in the system net with additional interesting features, for example a characteristic subset of the services of the hosted platform together with some of the pre- and postconditions of these services. The beholder of such an enhanced system net is provided with a complete view of important system activities without having to deal with the underlying complexity.[8]

[8] This is illustrated in the next section: The mobility system net is just a small part of the overall system consisting of several agents, platforms, technical substructure e.g. for the remote communication and so on. This is hidden from the user as long as the user does not wish to see the implementation details.

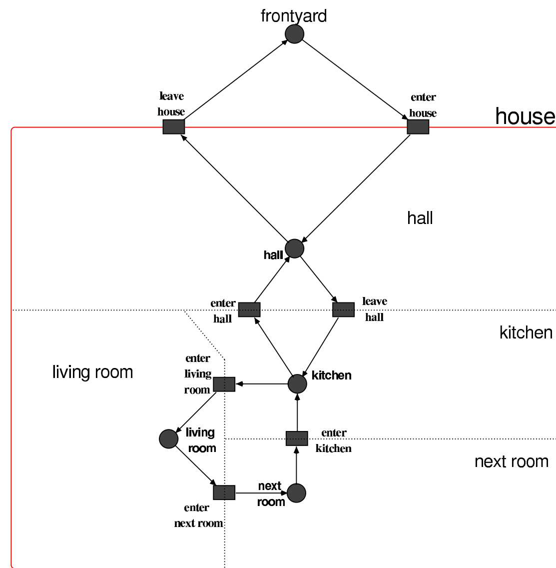


Fig. 9. Example system net

The difference between this proposal and a visualisation tool that shows some activities of e.g. a program running in background is twofold: First, using our proposal, the modelling process concludes with a running system model. A normal modelling process requires at least three stages to gain a comparable result: (a) model the system, (b) implement the model, and (c) write a visualisation for the program. Second, the visualisation of a system model at execution time is indeed the implementation of the system. This eliminates several potential sources of errors shifting from model to implementation to visualisation in an ordinary software design process.

4 Mobile Robot Case Study

To illustrate the modelling method introduced before, we introduce a small case study, a mobile household robot, unfortunately just a *software* bot.

4.1 Specification

The robot is internally implemented according to the Mulan architecture introduced in subsection 3.1. But that is not what the supervisor of the robot wants to know and see. What he requires is a simple view on his household, the robots location and state, and maybe some supplementary information. That is just what we provide with the extended system net view on the agent system.

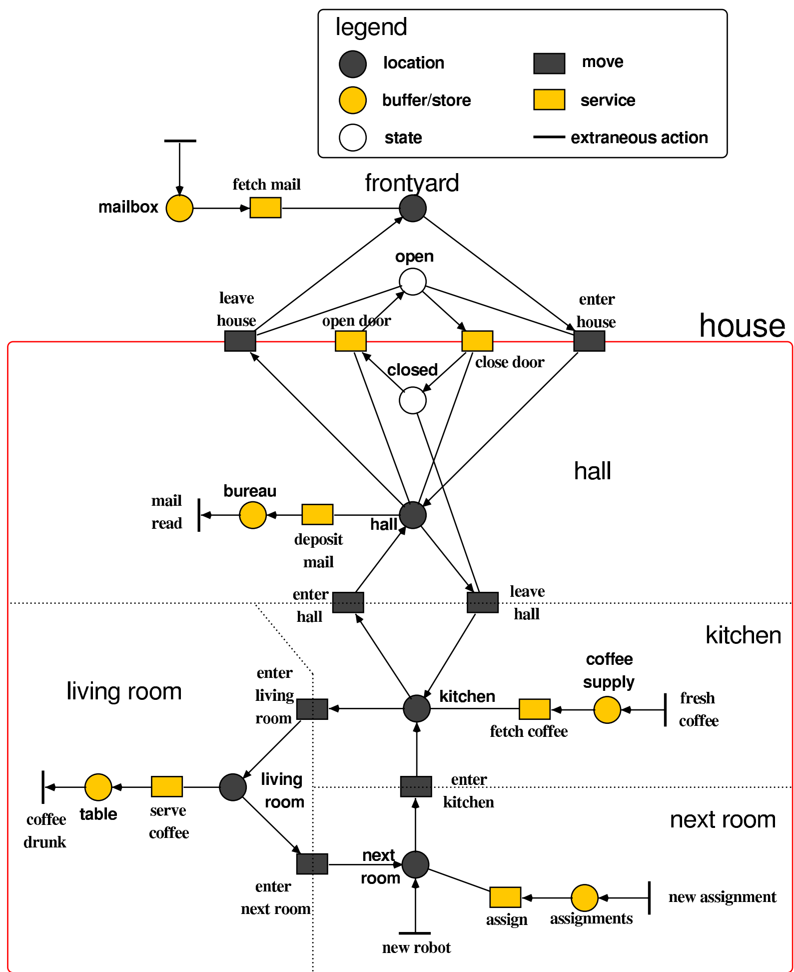


Fig. 10. Household System Net

The household is represented by the system net in Figure 10.[9] The household consists of several rooms (locations): hall, living room, kitchen, next room, and the front yard (dark places). Each room offers special services to the robot: it can fetch coffee in the kitchen, serve it in the living room, fetch mail in the front yard, open and close the door in the hall, and so on (light transitions). The possible movements from one location to another are displayed as dark transitions. Note that moving from room to room is not symmetric in this scenario. For example it is not possible to move directly from the kitchen to the next room. Service transitions are supplemented with additional information (service state/buffer, light places) showing for instance if new mail has arrived, coffee is available and so on. Extraneous actions not accessible for the robot are displayed as thin-lined transitions: arrival of new mail, new assignments for the robot etc.

The door of the house is used to show another possibility of viewing special parts of the system: the state of the door (open/closed) is directly modelled. This system state is not belonging to a single service (as for example the state of the mailbox), but is queried by a couple of service transitions including the movements into and out of the house.

This model of the household is filled with life by implementing an appropriate robot agent and defining the desired services for the platforms. The behaviour modelling for this kind of agents has been introduced elsewhere [KMR01].

Having defined the functionality of platforms and agents the parts are inserted in the household system net, which is a straightforward operation that claims for automation (tool support). After that the system is ready for execution.

4.2 Execution

Figure 11 shows an early state of a sample simulation of the agent system. For illustration purposes some nets that normally work "behind the scenes" are made visible in this screen shot. An ordinary simulation would just show the net house2 in the middle of the figure. It is outside the scope of this paper to explain the full functionality of the Mulan agent framework, so just the mobility overview net is regarded. [10]

Taking a closer view on the central household system net an additional feature becomes visible: Tokens can be visualised by an intuitive image. In the figure the incoming mail is visible in the upper left, and a fresh coffee on the lower right side. This is a one-to-one matching between token and image – whenever such a token is produced, moved, or deleted, the image follows. The case of the robot in the next room (lower middle) is a little bit more tricky (to implement), because what is really lying on the location place is a platform hosting the robot

[9] The use of colour greatly supports the differentiation of different types of places, transitions, or arcs. Unfortunately this is – even in the adapted form as in the figures – not so obvious in a black and white representation.

[10] Details of the Mulan framework are topic of an ongoing diploma thesis [Duv02] and a technical report [Röl02] as well as other papers, e.g. [DMR02]

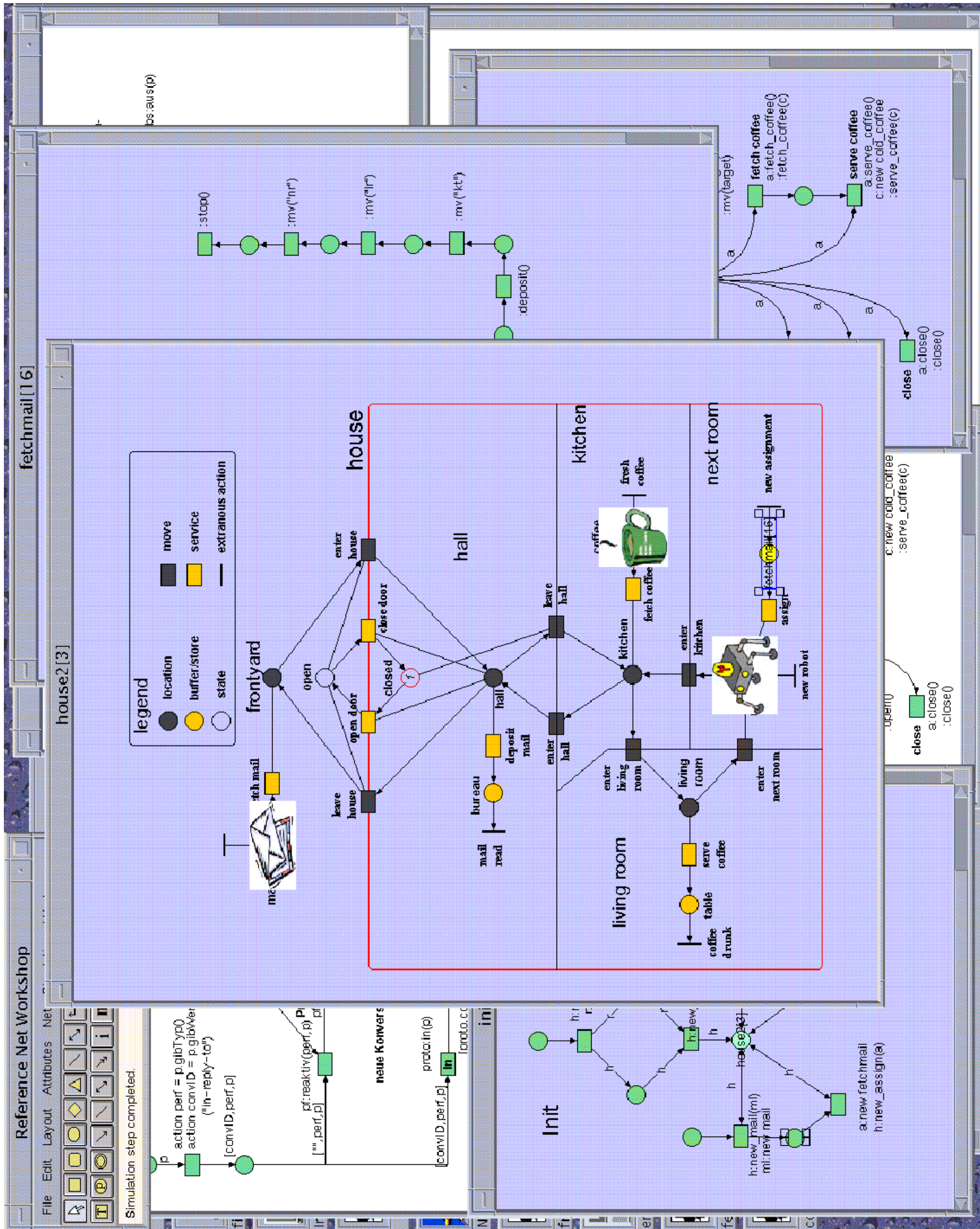


Fig. 11. System in Action

agent, and not the robot itself. What can be seen in the system net is actually a dynamic token figure of the platforms showing representatives of the agents hosted by the respective platform (empty platform means no image).

It is now easy to see the state of the overall system by looking at the system net. All the simplifications in presentation do not mean a loss of generality in the system's implementation. It is possible to take control of the whole agent system implementing the robot scenario by just double clicking on one of the tokens. The tool RENEW will then display the respective net (similar to those introduced in subsection 3.1), allowing for a complete inspection of the running system without having to interrupt.

In further execution steps the robot will receive one or more assignments, for example to serve fresh coffee in the living room. Its internal representation says that coffee is only available in the kitchen, so it moves there to look for the coffee service of the kitchen platform. In the scenario of Figure 11 this service is available, so the robot fetches the coffee and moves to the living room, where it serves the coffee. The robot ends the assignment by moving back to the next room and waiting for orders.

The behaviour modelling and planning the robot carries out to perform its tasks is outside the scope of this paper. The mobility system net helps greatly in validating a certain robot planning algorithm, for instance by unmasking bad habits of the robot like fetching the coffee, leaving the house for mail, returning to the living room later and serving cold coffee.

5 Conclusion and Outlook

In this article we have presented the paradigm of "nets within nets" for the modelling of mobility. Nets within nets are attractive, since the concept allows for an intuitive representation of mobile entities as well as an operational semantics which is implemented in the RENEW-simulator.

We have shown how this approach can help in modelling and executing mobile agent systems and underpinned our proposal with a small case study.

Besides the software-technical aspects, the exact semantics of nets within nets – in principal – allows for an analysis of mobile applications. This still needs more work, a starting point is [Köh02]. Formal treatment of mobile (agent) systems may prevent "classical" problems like deadlocks because of insufficient resources, but also helps in "modern" problems like security.

The use of an intuitive graphical formalism like Petri nets offers the room for a more widespread use as a modelling tool in agent oriented software engineering (AOSE) as opposed to text-based formalisms (mobile calculi), as the use of graphic modelling is an accepted technique in mainstream software engineering (see for example the Unified Modelling Language (UML) [Fow97]).

Additional formal results will be directly integrated in the design of the Petri net based multi agent system architecture MULAN.

References

[AB96] Andrea Asperti and Nadia Busi. Mobile Petri nets. Technical report, Department of computer science, University of Bologna, TR UBLCS-96-10, 1996.

[CGG99] Luca Cardelli, Andrew D. Gordon, and Giorgio Ghelli. Mobility types for mobile ambients. In *Proceedings of the ICALP'99*, volume 1644 of *LNCS*, pages 230–239. Springer-Verlag, 1999.

[CH94] Søren Christensen and Niels Damgaard Hansen. Coloured Petri nets extended with channels for synchronous communication. In Rober Valette, editor, *Application and Theory of Petri Nets 1994, Proc. of 15th Intern. Conf. Zaragoza, Spain, June 1994*, LNCS, pages 159–178, June 1994.

[COZ99] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. Modelling network topology and mobile agent interaction: An integrated framework. In *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC'99)*, 1999.

[DMR02] Michael Duvigneau, Daniel Moldt, and Heiko Rölke. Concurrent architecture for a multi-agent platform. In *Proceedings of the 2002 Workshop on Agent Oriented Software Engineering (AOSE'02)*, 2002.

[Duv02] Michael Duvigneau. A fipa-compliant multi agent platform. Master's thesis, Universiät Hamburg, 2002.

[Fow97] Martin Fowler. *Analysis patterns: reusable object models*. Addison-Wesley series in object-oriented software engineering. Addison-Wesley, 1997.

[HB01] Jarle Hulaas and Didier Buchs. An experiment with coordinated algebraic petri nets as formalism for modeling mobile agents. In *Workshop on Modelling of Objects, Components, and Agents (MOCA'01) / Daniel Moldt (Ed.)*, pages 73–84. DAIMI PB-553, Aarhus University, August 2001.

[Jen92] Kurt Jensen. *Coloured Petri nets, Basic Methods, Analysis Methods and Practical Use*, volume 1 of *EATCS monographs on theoretical computer science*. Springer-Verlag, 1992.

[KMR01] Michael Köhler, Daniel Moldt, and Heiko Rölke. Modeling the behaviour of Petri net agents. In J. M. Colom and M. Koutny, editors, *Proceedings of the 22st Conference on Application and Theory of Petri Nets*, volume 2075 of *LNCS*, pages 224–241. Springer-Verlag, June 2001.

[Köh02] Michael Köhler. Mobile object net systems: Petri nets as active tokens. Technical report, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2002.

[Kum98] Olaf Kummer. Simulating synchronous channels and net instances. In J. Desel, P. Kemper, E. Kindler, and A. Oberweis, editors, *Forschungsbericht Nr. 694: 5. Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 73–78. Universität Dortmund, Fachbereich Informatik, 1998.

[Kum02] Olaf Kummer. *Referenznetze*. Dissertation, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg, Deutschland, 2002.

[KW98] Olaf Kummer and Frank Wienberg. *Reference net workshop (Renew)*. Universität Hamburg, http://www.renew.de, 1998.

[KWD01] Olaf Kummer, Frank Wienberg, and Michael Duvigneau. *Renew - User Guide*. Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg, Deutschland, 1.5 edition, May 2001.

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

[Rei85] Wolfgang Reisig. *Petri Nets: An Introduction*. Springer-Verlag, Heidelberg, 1985.

[Röl02] Heiko Rölke. The Multi Agent Framework Mulan. Technical report, Universität Hamburg, 2002.

[Sun] Sunsoft. Java Online Reference Manual. `http://www.javasoft.com`.

[UTT00] Adelinde M. Uhrmacher, Petra Tyschler, and Dirk Tyschler. Modeling and simulation of mobile agents. *Elsevier, Artificial Intelligence?*, 2000.

[Val87] Rüdiger Valk. Modelling of task flow in systems of functional units. Technical Report FBI-HH-B-124/87, Universität Hamburg, 1987.

[Val98] Rüdiger Valk. Petri nets as token objects: An introduction to elementary object nets. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets*, volume 1420 of *LNCS*, pages 1–25, June 1998.

[Val00] Rüdiger Valk. Concurrency in communicating object Petri nets. In G. Agha, F. De Cindio, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, Lecture Notes in Computer Science, Berlin, 2000. Springer-Verlag.

[VC98] Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *ICCL Workshop: Internet Programming Languages*, pages 47–77, 1998.