Compositional Refinements In Multiple Blackboard Systems[1]

X. J. Chen
Scuola Normale Superiore
Piazza dei Cavalieri 7, 56100 Pisa, Italy

C. Montangero
Dipart. di Informatica, Univ. di Pisa
Corso Italia 40, 56100 Pisa, Italy

Abstract

In this paper we introduce CONESP, a concurrent system built according to the SMoLCS methodology to provide an abstract model of the coordination language Extended Shared Prolog (ESP). ESP is based on the integration of the blackboard paradigm with Logic Programming. CONESP is a hierarchy of entities, each consisting of a passive blackboard tree and a collection of active components including parallel agents and dynamic (sub) entities. An implementation relation between two hierarchies is defined, which is shown to be compositional.

ESP is being used in the Oikos environment for software process modeling. The results of this paper are the base for the formal verification of the correctness of the software process models built by stepwise-refinements in Oikos.

1 Introduction

This paper presents the first results towards an algebraic treatment of software process specifications in Oikos [1], and of the related refinement method.

Software process modeling deals with the problem of describing the entire life cycle of software production. Among the current approaches, Darwin [11] has a declarative approach, introducing Law Ruled environments; Marvel [9] models tool activation with a rule based language, emphasizing automatic planning; Arcadia [14] is a very comprehensive approach to software process enactment, resting on the extension of Ada with a relational calculus; Melmac [8] introduces an intermediate level representation of software processes by high-level Petri Nets.

Oikos is an environment for software process modeling which integrates the blackboard paradigm with Logic Programming in a hierarchical system. In fact, the blackboard paradigm is suitable to deal with the problems related to the cooperation and coordination among people and machines, and Logic Programming allows to specify software processes declaratively. An essential feature of Oikos is that process specifications are developed by step-wise refinements: the hierarchy is well suited to such an approach. In Oikos, all the software process modeling entities, like processes, environments, roles, services, etc. are represented by a blackboard system which is dynamically organized into a hierarchy.

[1]This work has been partially founded by Progetto Finalizzatto Sistemi Informatici e Calcolo Parallelo, Sotto Progetto 6 under contract 91.00920.PF69.

Oikos exploits the logic distributed language Extended Shared Prolog [1,6] to enact, i.e. execute, software processes. As the name shows, ESP extends Shared Prolog [4]: this paper gives a formal specification of ESP extending the one given for Shared Prolog. In doing so, we are taking into account the need of proving the correctness of the refinements that are essential in the Oikos approach. Therefore, our formal description accomodates smoothly a notion of refinement, i.e. how an ESP system can implement another one (in general, with more details and with a higher degree of parallelism).

To cover the characteristics of ESP we introduce CONESP, a concurrent system constructed using labelled transition system according to the Structured Monitored Linear Concurrent Systems (SMoLCS) methodology [2,13]. SMoLCS is a methodology for the formal specification of concurrent systems and languages, especially useful in the formal description of large complex systems with multilevel architecture and interference among the sequential part and the concurrent part.

The paper is organized in this way: section 2 and 3 review ESP and SMoLCS; section 4 describes CONESP and section 5 defines the implementation relation in ESP, with the proof that it is compositional.

2 Shared Prolog and ESP

Shared Prolog and ESP are based on the blackboard model of problem solving. According to this model, the knowledge on a particular problem is partitioned into distinct subsets, in order to keep domain knowledge separated from control knowledge and to organize communications via a centralized data structure, named blackboard.

2.1 Shared Prolog

A Shared Prolog system is composed of a unique blackboard and of a collection of agents working in parallel and communicating via the blackboard. Neither a global state nor a global clock must be assumed.

The behavior of an agent is defined by a theory, which is a Prolog program extended by a guard mechanism coordinating communication and synchronization via the blackboard. More precisely, a theory is a Prolog program augmented with a set of activation patterns, each having form:

{In_Guard} Read_Guard | Body {Out_Set}

which specifies: (i) a set of atoms to be consumed before the theory can be activated (In_Guard); (ii) some conditions to be verified on the blackboard before the theory can be activated (Read_Guard); (iii) the initial goal of the logic program of the theory (Body); (iv) a set of atoms that will be written on the blackboard at the end of the activation (Out_Set).

A blackboard holds a multiset of facts which are Prolog atoms. Getting/putting facts exploits unification. Input and output on the blackboard are mutually exclusive.

2.2 ESP

An ESP system has a tree of blackboards. Each node on this hierarchy is a blackboard containing agents, facts, and subsystems. Each agent belongs to a blackboard, i.e. its source of facts, and has a list of targets to put its facts.

ESP patterns extend Shared Prolog patterns in three ways: First, there are several Out_Sets, each followed by a target (denoted by symbol "@", the target of the agent's own blackboard may be omitted); second, the failure of execution of Prolog program may also cause output. So, the output is divided in two: a success Out_Set and a failure Out_Set, separated by ";". The failure Out_Set is optional, and if there's no failure Out_Set when the Prolog execution fails, nothing will be written on the blackboards. Finally, there may be several *read guards* and *in guards* in a pattern. Generally, an ESP pattern has the following form:

$$\begin{array}{lr}
read_1 \ \{in_1\} \ \ldots \ read_n \ \{in_n\} & n > 0 \\
\mid \ \text{body} & \\
\{succ_1\}[@target_1]\ldots\{succ_m\}[@target_m] & m > 0 \\
[; \ \{fail_1\}[@target_1]\ldots\{fail_p\}[@target_p] \ ] & p > 0
\end{array}$$

Example *SeqImp* in figure 1 gives the initial state of the ESP program in table 1, which models a Sequential Machine:

```
theory coord(Target2,Target3) :-
    {input(X), state(Y)}
    |
    {input1(X), input2(Y)}@Target3
#
    {out(X)}
    |
    {state(X)}
    {output(X)}@Target2
```

```
theory pcomb(Target) :-
    {input1(X), input2(Y)}
    |
    f(X,Y,Z).
    {out(Z)}@Target
with
    f(X,Y,Z):- computes Z from X, Y.
```

Table 1. The theories of *SeqImp*

There are two blackboards (represented by boxes with their names on up-left corners) named *Seq* and *Comb* respectively.

Blackboard *Comb* and the agent on it (working on theory *pcomb* in Table 1) model the combinational network of the Sequential Machine, while blackboard *Seq* and the agent on it (working on theory *coord*) model the rest of the machine: inputs are sent to *Seq*, and the agent on *Seq* feeds them to *Comb* together with the current state, receives the output, updates the state and produces the output of the Sequential Machine.

Blackboard *Seq* contains a fact *state(initial)*, and an agent (denoted by a circle) with three targets (denoted by arrows from circles to blackboards) for its output: its own blackboard *Seq* (denoted by number 1), the outer system (2) and blackboard *Comb* (3). This agent works on theory *coord* which uses parameter *Target2*/*Target3* as its second/third target. In this example, they are instantiated as *UP*/*Comb* (Note: *UP*

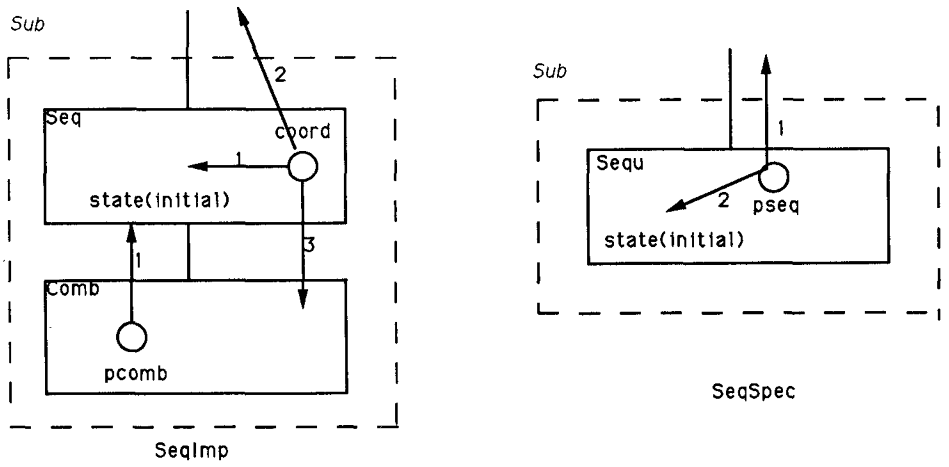

Figure 1: Two specifications of the Sequential Machine

is a special blackboard name for all outside targets). *Target*1 is omitted since it is the agent's own blackboard. Theory *coord* has two patterns (separated by #). In the first pattern, it consumes facts that match *input*(X) and *state*(Y) respectively, and then gives facts *input*1(X) and *input*2(Y) to its third target *Target*3. In the second pattern, it consumes facts that match *out*(X) and then gives facts like *state*(X) to its own blackboard and facts like *output*(X) to its second target *Target*2. In both these patterns, the *Body* is empty.

Initially, blackboard *Comb* contains no facts but only an agent working on theory *pcomb*, which has only one pattern and one target which is instantiated to blackboard *Seq*. This pattern consumes facts of form *input*1(X) and *input*2(Y), executes goal $f(X,Y,Z)$ by the Prolog program (beginning from keyword *with* in the theory), and then gives output *out*(Z) to its target. □

The blackboard hierarchy is dynamic, i.e. new subtrees of blackboards may be created at run-time. At the same time, the targets of the agents are also dynamically defined. So the total dynamicity of the system means: the agents, the sources from where to absorb information and the targets to where to send information may change. Due to the lack of space, we no longer deal with the dynamic facets of ESP.

2.3 Refinements in ESP

A software process model may be very large. To deal with it, one may construct, at first, a small system with a simple structure only to describe the main properties (the tasks to fulfill), then substitute it with another one with more implementation details. For example, consider *SeqSpec* in figure 1, where theory *pseq* is

theory $pseq(Target) :-$
$\quad\{input(X), state(Y)\}$
$\quad|$

$f(X,Y,Z)$.
$\{output(Z)\}@Target$
$\{state(Z)\}$

with

$f(X,Y,Z)$:- the same as that in theory *pcomb*.

It can be seen as the initial specification of the Sequential Machine corresponding to *SeqImp* in figure 1. So one may use *SeqSpec* at the beginning and then substitute it by *SeqImp*.

The problem rises: is this substitution correct? I.e., is *SeqImp* a correct implementation of *SeqSpec*, which means: can *SeqImp* generate all I/O streams that can be generated by *SeqSpec*, and will it not generate spurious streams in the context of the substitution? The way to solve the problem is to forget intermediate derivations, which is done by the monitoring step in CONESP that gives the support to define the implementation relation in CONESP.

3 A Brief Overview of SMoLCS

To build CONESP, which is the formal system that defines ESP semantics, we adopt the SMoLCS methodology [2,3,13] which is based on labelled transition system. A set of processes is described by a labelled transition system, in which *states* represent the states of processes, and *transitions* represent the capability of the processes to evolve from one state to another while the interaction with the outside world is represented by the label. In other words, the label contains the information for the condition of the external environment so that the capability of the processes to evolve becomes effective. This interpretation of labelled transition system has turned to be classical since it was inspired by CCS [10] and SOS [12]. In the simplest case, when the transition is purely internal to the system and there is no relationship with the environment, the label can be dropped or better represented by a special label.

Based on labelled transition system, SMoLCS is a methodology intended to add abstraction and modularity to the specification of concurrent systems and languages, especially considered for the formal description of large systems.

Def. **A labelled transition system is a triple** $(STATE, LABEL, \Longrightarrow)$ where $STATE/LABEL$ is a set of *states/labels* of the system, and $\Longrightarrow \subseteq STATE \times LABEL \times STATE$ is the transition relation.

Note Usually, we simply write $s \overset{l}{\Longrightarrow} s'$ if $(s,l,s') \in \Longrightarrow$.

Example If agent $Agent_1$ can put facts F onto blackboard B, and then changes into $Agent_1'$, this can be represented by

$$Agent_1 \xRightarrow{Send(B,F)} Agent_1'$$

□

Given transition systems for the simple components, the dynamic activity of a compound system is defined in SMoLCS in a canonical way, following three steps: syncronization, parallelism and monitoring, which are specified by giving appropriate abstract data types. In this way, one can easily modify part of a specification, which is badly needed in practice for large projects. To fix notations, in the following we review these steps, which are applied in the next section to define CONESP.

The syncronization step defines those transitions that represent syncronized actions of a group of process components and their effects on global information: given a transition system CTS (with transition relation $\Longrightarrow$) that represents the process components, the syncronization step defines a new algebraic transition system STS where the transition relation $\longrightarrow$ corresponds to the syncronized actions of a group of process components.

The state of STS is a couple whose elements are a multiset of states in CTS and some values (called global information of the process components) representing the status of the passive components. Here, passive means that it has no transitions of its own, but may be changed as a consequence of the transitions of processes. Thus, the states in STS have form

$$< g, pr_1 \mid \ldots \mid pr_n >$$

where g is the global (passive) part, pr_i are states of CTS, and $pr_1 \mid \ldots \mid pr_n$ is a multiset of (parallely composed) process states in CTS.

The transitions of STS are deduced from the transitions of CTS by STS axioms that have the following form:

$$\bigwedge_{1 \leq j \leq n} pr_j \overset{l_j}{\Longrightarrow} pr_j' \ \wedge \ Cond \ \supset \ < g, pr_1 \mid \ldots \mid pr_n > \xrightarrow{sl} < g', pr_1' \mid \ldots \mid pr_n' >$$

where sl is a new label denoting the interaction of $pr_1, \ldots, pr_n$ with the outside, and $Cond$ expresses that the actions of process components pr_i $(i = 1, ..., n)$ can be syncronized. The intuitive meaning of such an axiom is that, if each pr_i can evolve into the new process state pr_i' by performing the action l_i, and $Cond$ holds, then the whole system can evolve into the new state $< g', pr_1' \mid \ldots \mid pr_n' >$ by some label sl, while the global information g is relatively changed into a new one w.r.t. the actions $l_1, \ldots, l_n$.

The parallelism step defines those transitions that represent the admissible parallel executions of a group of syncronized actions and their complex transformation of global information. For example, the problem of mutual exclusion is faced on this step.

The actions on this step are defined by using a composition function $\|$ (binary, commutative and associative) on actions of a group of process components. Intuitively, it defines that two actions can be executed in parallel (without syncronization). The actions that need to be considered for the composition are the syncronized actions defined in the syncronization step and those already obtained by composition.

The following axiom expresses the capability of two actions to be executed in parallel:

$$< g, ms_1 > \xrightarrow{sl_1} < g_1, ms_1' > \ \wedge \ < g, ms_2 > \xrightarrow{sl_2} < g_2, ms_2' > \ \wedge \ Cond \ \supset$$
$$< g, ms_1 \mid ms_2 > \xrightarrow{sl_1 \| sl_2} < g', ms_1' \mid ms_2' >$$

where *Cond* defines the transformation of the global information (from g to g' as a function of g and g') caused by action $sl_1 \parallel sl_2$.

The Monitoring step may define some global restrictions on the transitions of the entire system, like restricting the labels which can be observed from the outside.

Analogously to the previous steps, the monitoring step is also defined by giving a new (modular) concurrent system starting from the concurrent system obtained in parallelism step. The axioms on this step offer some conditions under which the action in parallelism step

$$< g, mp > \xrightarrow{pl} < g', mp' >$$

is allowed by the monitor and it will become an action of this step represented by transition

$$< g, mp \mid mp_1 > \xrightarrow{l} < g', mp' \mid mp_1 >$$

where mp_1 represents the multiset of the states of the active components in previous step, and l represents the interaction of the resulting action with the external environment.

4 CONESP, a concurrent system for ESP

In this section, we describe an ESP-entity as an abstract data type and then introduce the derivations that define its behaviors.

4.1 CONESP Entities

CONESP is constructed as a hierarchy of ESP-entities, where each ESP-entity corresponds to a blackboard system in ESP. An ESP-entity named $NAME$ can be expressed as:

$$NAME :< TR, AC >$$

where $< TR, AC >$ represents the state of the entity: a passive blackboard tree TR and a collection AC of parallel active components (separated by "|").

4.1.1 TR: the blackboard tree

The passive part TR represents the blackboard tree of ESP and is a dynamic structure describing the evolving state of the dynamic system. Its nodes are multisets of facts. Each blackboard in the tree has a unique name. This passive part is specified as an abstract data type TREE using partial algebra [7]. We give here only a simple example.

Example 4.1.1 The passive part of the ESP-entity representing the blackboard tree of $SeqImp$ in figure 1 can be specified as

$$(R(Seq, \{state(initial)\}), Seq, Comb, \phi) \qquad (Trimp)$$

where ϕ is the empty multiset of facts. In general, (TR, B_1, B_2, F) is a tree generated from tree TR by adding a new blackboard B_2 under blackboard B_1, with the initial set of facts F. Also, $R(B, F)$ denotes a tree with only a root B containing facts F.

4.1.2 AC: a collection of active components

There are two kinds of active components: the agents and (sub) ESP-entities.

Each agent in an entity has a coordinate blackboard, denoting the source to get facts. In order to meet the characteristics in ESP that each agent has a list of targets to give its output, we also explicitly express, for each agent, this dynamic binding as a list of targets. Besides, each agent has a predefined theory, the task it executes. In this sense, we explicitly specify an agent in CONESP together with these three elements as prefixes. Generally, we express an agent as

$$(B, Th, Tar) : Astate$$

where B is the blackboard of the agent, Th is its theory, Tar is its target list, $Astate$ is a state holding execution information and is discussed in section 4.2.1.

Note The target list of an agent includes parameters and may be changed dynamically.

Example 4.1.2.1 The agents in *SeqImp* (figure 1) are expressed as

$$(Seq, coord, [Seq, UP, Comb]) : nil \qquad (Agimp1)$$

$$(Comb, pcomb, [Seq]) : nil \qquad (Agimp2)$$

Note: Here, *nil* is the state, in which an agent is ready to activate the patterns in its theory. □

The detailed structure of an ESP-entity is shown in figure 2.

Example 4.1.2.2 The initial entity *SeqImp* in figure 1 (named *Sub*) can be described as

$$Sub :< Trimp, Agimp1 \mid Agimp2 >. \qquad (SeqImp)$$

Example 4.1.2.3 The entity named *UpName* (figure 3, right) contains entity *SeqImp* and some other active components *UpAc*, and is expressed as

$$UpName :< UpTr, UpAc \mid SeqImp >$$

where *UpTr* is its passive part.

4.2 CONESP Specifications

The structure of CONESP specifications is as follows: (i) Agent Transition describes how the agents work on their theories. (ii) Syncronization Step describes the syncronized transitions of entities including the syncronization among agents, blackboards, and subentities. (iii) Parallelism Step describes the entity transitions on all agents and subentities. On this step, conditions w.r.t. ESP, e.g. mutual exclusion on blackboard,

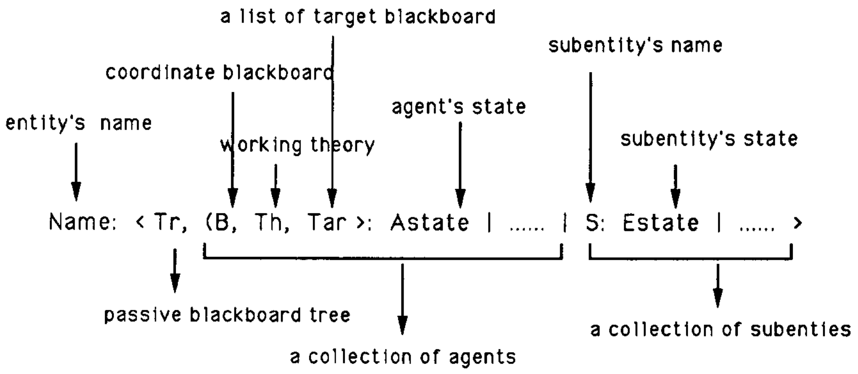

Figure 2: structure of entities in CONESP

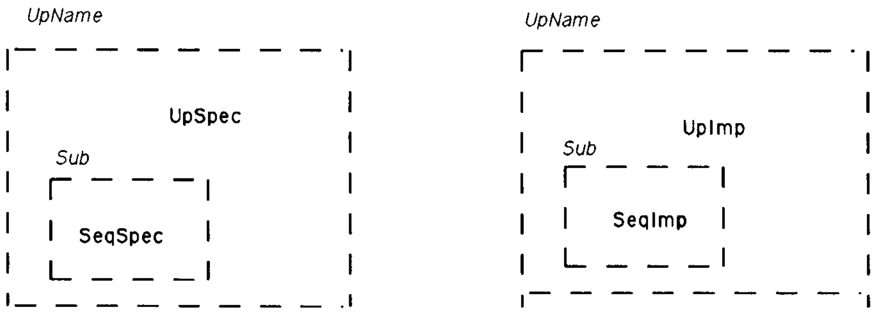

Figure 3: ESP-entity examples

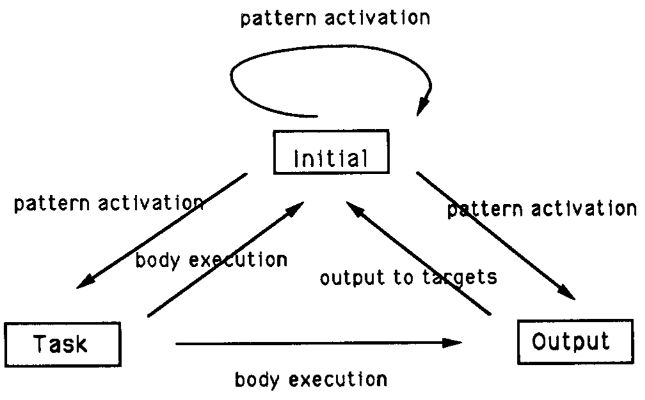

Figure 4: execution states of theories in CONESP

will be given. (iv) Observational Step describes the observable actions and related transitions.

We use the following sorts for the axiom quantifiers: *bid* for blackboard identifier and subentity's names, *facts* for multiset of Prolog atoms (sort *fact*) in a blackboard, *th* for theory's names, *name* for entity's names, *sub* for substitutions, *tree* for passive blackboard trees, *tar* for lists of agent's targets, *ag/ent* for agents/entities and *ac* for active components (whenever it is convevient we will distinguish between agents and entities).

4.2.1 Agent Transition (theory execution)

Like in Shared Prolog, each agent works recursively on its theory, having three execution states as shown in figure 4:

1. It may stay in state *Initial*. In this state, the agent is ready for pattern activation, i.e. to choose, among those in its theory, a pattern whose guards are satisfiable.

2. It may stay in state *Task*, ready to do the task of this activated pattern, e.g. *Prolog*($Body\theta$), which indicates that the agent is going to execute Prolog program for goal $Body\theta$. Here *Body* is the body part of the pattern chosen in the previous step, and θ is the substitution got from the previous step. Such *Tasks* are described as internal derivations, i.e. we do not go into the details of Prolog execution. Other tasks include creating subentities, deleting subentities, instantiating targets, deleting targets and are not dealt with here, due to the lack of space.

3. After finishing a task, the agent is in state *Output*, and is going to give output to its targets. This output comes from the Out_Set of the pattern together with

the substitution from pattern activation in the first step and, if it has successfully terminated the body, the resetting substitution.

Generally, we use *nil* for state *Initial*; *A.nil* for state *Output* where A is the output set denoted by ε if it is empty; $A.(B;C).nil$ for the state *Task* where A is the task to fulfill and B/C is the success/failure output set.

Note Since the body part and postactivation part in ESP theory may be empty, we may also go from state *Initial* to state *Output* or to itself, from *Task* to *Initial*. □

The change from one state to another should satisfy the corresponding rules on agent transitions including pattern activation, body execution, output to targets. Generally, we have the following schema for pattern activation:

$\forall\ B : bid,\ T : tar,\ \theta : sub$

$$(B, Th, T) : nil \xrightarrow{Activate(B, I_{Th,Pa}, R_{Th,Pa}, \theta)}$$

$$(B, Th, T) : Prolog(Goal_{Th,Pa}\theta).(Succ_{Th,Pa}\theta; Fail_{Th,Pa}\theta).nil$$

for each pattern Pa in theory Th such that its body is a Prolog goal $Goal_{Th,Pa}$.

Note

1. CONESP specification is parameterized and this is only a schema: for any theory Th in the program, it may be applied to any of its pattern Pa provided that its body part is a Prolog goal.

2. $Prolog(F)$ calls for Prolog program using F (including variables) as a goal. $I_{Th,Pa}/R_{Th,Pa}$ is the In_Guard/Read_Guard of pattern Pa in theory Th.

3. $Succ_{Th,Pa}/Fail_{Th,Pa}$ has form $AddF(target_1, out_1) \parallel \ldots \parallel AddF(target_m, out_m)$ corresponding to the success/failure Out_Set (cfr. section 2.2) of pattern Pa in theory Th. $AddF(target_i, F)$ adds a set of facts F to the i-th target of the agent.

4. Blackboard B and target T are parameters for the coordinate blackboard in which this theory is executed, and the instantiation for the targets of this theory. Substitution θ appears also as a parameter. Its value will come from the communication with its blackboard by matching $I_{Th,Pa}$ and $R_{Th,Pa}$ in blackboard B (See next step for the communication between agents and their blackboards).

As a special case when the Body part of a pattern is empty, the agent goes directly to give the success Out_Set after activating the pattern.

Example 4.2.1 Theory *coord*'s first pattern (section 2.2) has an empty body part. For agent *Agimp*1 (Example 4.1.2.1) in entity *SeqImp* (figure 1), the agent's blackboard and targets being instantiated by *Seq*, *Seq*, *UP* and *Comb* respectively, we have

$\forall\ \theta : sub$

$$Agimp1 \xrightarrow{Activate(Seq, \{state(X), input(Y)\}, \varepsilon, \theta)}$$

$$(Seq, coord, [Seq, UP, Comb]) : AddF(3, \{input1(X), input2(Y)\}\theta).nil \qquad (Agimp1')$$

4.2.2 Syncronization Step

An agent's action includes creating/deleting subentities, pattern activation (getting facts), internal derivation (those that have no effects on other part of the entity, e.g. instantiating/deleting targets, executing Prolog program), sending facts to targets (blackboards or subentities).

An entity may receive facts from an agent from outside. As we hide all the details of an entity to outside, the agent knows neither the structure of the entity nor any blackboard name in it. For the moment, we adopt the principle that the agent from outside sends facts to an entity by calling its name, and the facts are put into the root of the entity. We use $Send(S, F)$ to describe the agent's action of sending facts F to entity S, and we use $In(S, F)$ to describe that the entity receives facts from outside. On the other hand, an entity may send facts F into a blackboard B in super entity, which is described as $Out(B, F)$. Thus, as an active component, an entity acts in two ways: $In(S, F)$ and $Out(B, F)$. These two actions are distinct from the others.

On this step, we define entity transitions of syncronized actions and their effects to the blackboard tree. The rules for syncronizations considering these actions have the canonical form in the same step in SMoLCS (cfr. section 3). Notice that in blackboard systems, no two agents can communicate directly, but via the blackboard. So the syncronization happens only between agent and blackboard, between subentity and blackboard, and between agent and subentity. Here we only show how the agent's pattern activation effects its blackboard and causes the transition of its entity.

When an agent is in *Initial* state, it may activate a pattern in its theory provided that In_Guard and Read_Guard of this pattern can be satisfied by the current state. This test is done by

$$eval(In\_Guard, Read\_Guard, Blackboard1) =< Blackboard2, \theta >$$

where function *eval* takes the In_Guard, the Read_Guard, the blackboard of the system (*Blackboard*1), and returns, if the evaluation succeeds, the set of atoms (*Blackboard*2) to be retracted from blackboard (*Blackboard*1) and the computed substitution (θ) to be passed to the rest of the pattern. Otherwise, the special symbol *FAILURE* is returned (cfr. [4]).

From section 4.2.1, we know that in $Agent \xRightarrow{Activate(B,I,R,\theta)} Agent'$, the agent has coordinate blackboard B, the activated pattern has In_Guard I, Read_Guard R, and the matching between the guards and facts in blackboard B should be θ. So we have

$$\forall\ Tr : tree,\ Pr, Pr' : ac,\ B : bid,\ I, R, F : facts,\ Ename : name, \theta : sub$$

$$Pr \xRightarrow{Activate(B,I,R,\theta)} Pr' \wedge eval(I, R, getb(Tr, B)) =< F, \theta > \wedge D(< Tr, Pr >) \supset$$

$$Ename :< Tr, Pr > \xrightarrow{Rec(B,F)} Ename :< delf(Tr, B, F), Pr' >$$

for the entity transition of theory activation, where the matched facts F are deleted ($delf(Tr, B, F)$) from the blackboard B in the tree Tr.

Note Partial algebras [5] are used. Here $D(< Tr, Pr >)$ denotes the definedness of entity $Ename :< Tr, Pr >$. Condition $eval(I, R, getb(Tr, B)) =< F, \theta >$ guarantees the definedness of $< delf(Tr, B, F), Pr' >$ since we have that $D(< Tr, Pr >)$. *getb* is a function that, given a tree and a blackboard name in the tree, returns the multiset of facts on the named blackboard in the tree.

Example 4.2.2 Once entity *SeqImp* (Example 4.1.2.2) receives a fact *input*(a) from outside into its blackboard *Seq*, it transforms into $SeqImp_1$, i.e.

$$Sub :< Trimp_1, Agimpl \mid Agimp2 >.$$

where $Trimp_1 = addf(Trimp, Seq, \{input(a)\})$. The transition in which *Agimpl* activates the first pattern of theory *coord* (Example 4.2.1), can be used in entity $SeqImp_1$, because both *state*(X) and *input*(Y) can be matched on blackboard *Seq* in the tree $Trimp_1$. In other words,

$$eval(\{state(X), input(Y)\}, \varepsilon, getb(Trimp_1, Seq)) =< F, \theta_0 >$$

is true with

$$F = \{state(initial),\ input(a)\} \quad \theta_0 = \{X \leftarrow initial,\ Y \leftarrow a\}$$

So we have

$$Sub :< Trimp_1, Agimpl > \xrightarrow{Rec(Seq,\{state(initial),\ input(a)\})} Sub :< Trimp_2, Agimpl' >$$

where $Trimp_2 = delf(Trimp_1, Seq, \{state(initial), input(a)\})$

4.2.3 Parallelism Step

According to SMoLCS, we describe here how the transitions introduced in the previous step may happen in parallel, on all active components. The essence of this step is to give conditions under which if entity $Entity_1$ may transit into $Entity_2$ by $Act_1 \parallel \ldots \parallel Act_n$, and may transit into $Entity_3$ by Act_{n+1}, then it may transit into $Entity_4$ by $Act_1 \parallel \ldots \parallel Act_n \parallel Act_{n+1}$.

From ESP, we know that no two accesses to the same blackboard are allowed at a time. This also satisfies the condition in Shared Prolog that among several patterns of a theory, only one can succeed at a time. So, if we want to put an action *Sact* on syncronization step into an action *Pact* on parallelism step, we must test if *Sact* is going to access any blackboard already used in *Pact*.

Example 4.2.3 It is always possible to put the internal action τ into any transition on parallelism step, without any condition. For example, suppose the τ action is executed by a collection of active components Prs_1

$$Ename :< Tr, Prs_1 > \xrightarrow{\tau} Ename :< Tr, Prs_1' >$$

and suppose a collection of active components Prs_2 may change into Prs_2' by action $Rec(B, F)$ which changes the blackboard tree Tr into Tr':

$$Ename :< Tr, Prs_2 > \xrightarrow{Rec(B,F)} Ename :< Tr', Prs_2' >$$

then without any condition, we have

$$Ename :< Tr, Prs_1 \mid Prs_2 > \xrightarrow{Rec(B,F)} Ename :< Tr', Prs_1' \mid Prs_2' >$$

4.2.4 Monitoring Step

In section 4.2.3, we have seen that an entity has two kinds of actions for the communication with the outside. On this step, function *Visible* is introduced to select out these actions from those in parallelism step, leaving all the others denoted by τ, e.g.

$$Visible(Out(B_1, F_1) \parallel Rec(B_2, F_2)) = Out(B_1, F_1)$$

Besides, we add a restriction on this step that no two output actions of an entity can succeed at a time, i.e.

$$A \xrightarrow{Out(B,F)\|Other} A'$$

on the parallelism step, implies

$$A \xrightarrow{Out(B,F)} A'$$

on this monitoring step, provided that there's no other action $Out(B', F')$ appearing in *Other*.

From the previous two steps, we know that action

$$In(S_1, F_1) \parallel \ldots \parallel In(S_n, F_n) \text{ for } n > 1$$

is not allowed because this means, according to our principle of the communications between entity and subentity (section 4.2.2), to access the root of the blackboard tree at the same time, which is not allowed according to the mutual exclusion on blackboard.

As a consequence, we have only four kinds of actions on this step ($F, F' : facts$, $B : bid$, $S : name$):

$$\tau,\ In(S,F),\ Out(B,F),\ In(S,F) \parallel Out(B,F')$$

5 Implementation Relation

Based on the monitoring step, we turn to the sequences of the actions defined on monitoring step in order to give a suitable notion of implementation relation in ESP, as announced in section 2.3.

We want to consider action sequences (of sort *seq*) that start from a given state, but not all of them. For the initial state *SeqImp*, for example, we expect that the inputs are unifiable to *input*(X) or *state*(X) (which are all that make sense for *SeqSpec*), but not to *out*(X).

Def. Action sequence $l_1 \ldots l_n$ ***is covered by*** X, denoted by $l_1 \ldots l_n \sqsubset X$, if

l_i is $In(S,F)$ or $In(S,F) \parallel Out(B,F')$ implies $F \subseteq X$. □

Notice that F' can be any set of facts, i.e. only the input set is concerned.

Function *IIF* is thus introduced to get from a given entity the interesting input facts under our consideration. For instance, *IIF*(*SeqSpec*) contains all the facts unifiable to *input*(X) or *state*(X), while *IIF*(*SeqImp*) contains all the facts unifiable to *input*(X) or *state*(X) or *out*(X). So the sequences used to compare *SeqSpec* and *SeqImp* should be covered by *IIF*(*SeqSpec*).

Observing predicate $Perform(\sigma, A, X)$ is used to characterize our observations — the action sequence σ starting from the initial state A and covered by set of facts X. By definition,

$$Perform(l_1 \ldots l_n, A, X) \equiv \bigwedge_{i=1}^{n} A_i \xrightarrow{l_i} A_{i+1} \ \wedge \ l_1 \ldots l_n \sqsubset X$$

Now we introduce our implementation relation, denoted by $\leadsto$. $A \leadsto B$ means that B is an implementation of A:

Def. $A \leadsto B$ iff $IIF(A) \subseteq IIF(B)$ and

$\forall \sigma : seq.\ Perform(\sigma, A, IIF(A))$ iff $Perform(\sigma, B, IIF(A))$.

Example Now our problem on Sequential Machine can be stated as

$\forall \sigma : seq.$

$Perform(\sigma, SeqSpec, IIF(SeqSpec)) \equiv Perform(\sigma, SeqImp, IIF(SeqSpec))$? □

It is easy to see that if $X_1 \subseteq X_2$ then

$Perform(\sigma, A, X_2)$ implies $Perform(\sigma, A, X_1)$.

So we have the following

Fact $\leadsto$ is transitive.

Lemma If A_1 and B_1 have the same name S, $A_1 \leadsto B_1$ and agents in R only send facts in $IIF(A_1)$ to S (i.e. in their actions $Send(S, F)$, F always belong to $IIF(A_1)$), then

$$EA_1 = N :< Tr_1, A_1 \mid R > \leadsto EB_1 = N :< Tr_1, B_1 \mid R >$$

Proof We prove for the universe U of facts,

$Perform(\sigma, EA_1, U)$ implies $Perform(\sigma, EB_1, U)$

The proof of the other direction is analogous.

Consider any $l_1 \ldots l_n : seq$, s.t. for $1 \leq i \leq n$, we have $EA_i \xrightarrow{l_i} EA_{i+1}$ where $EA_{i+1} = N :< Tr_{i+1}, A_{i+1} \mid R_{i+1} >$ and $A_i \xrightarrow{l_i^A} A_{i+1}$.

Now, since the environment behaves correctly, i.e. agents in R_i only send facts in $IIF(A_1)$ to S, we have $l_1^A \ldots l_n^A \sqsubset IIF(A_1)$, and since $A_1 \leadsto B_1$, we can find B_i s.t. $B_i \xrightarrow{l_i^A} B_{i+1}$. Given now

$$EB_i = N :< Tr_i, B_i \mid R_i > \text{ and } EB_{i+1} = N :< Tr_{i+1}, B_{i+1} \mid R_{i+1} >,$$

we have

$$EB_i \xrightarrow{l_i} EB_{i+1}$$

because we only need to consider syncronization between S and the environment and, according to CONESP specifications, the conditions for syncronization do not include information in the entities (e.g. A_i, B_i), but the information in the syncronized actions (e.g. l_i^A). □

The composibility of $\leadsto$ may be stated as

Theorem If $A_i \rightsquigarrow B_i$ (i=1,2,...,n), where A_i, B_i have name S_i, agents in R only use facts in $IIF(A_i)$ in their actions $Send(S_i, F)$, and

$$E = N :< Tr, A_1 \mid \ldots \mid A_n \mid R > \rightsquigarrow N :< Tr', A_1 \mid \ldots \mid A_n \mid R' >$$

then $E \rightsquigarrow N :< Tr', B_1 \mid \ldots \mid B_n \mid R' >$

Proof immediate by Lemma and transitivity. □

This composition theorem protects the implementation relation when subentities are substituted by their implementation. So we may solve, for instance, the problem shown in figure 3. That is: if we substitute a subentity (*SeqSpec*) by its implementation (*SeqImp*) which has more implementation details, the total entity (*UpSpec*) is also changed into its implementation (*UpImp*).

6 Future Work

We are now working on proof techniques. In the basic cases, the proof that $A \rightsquigarrow B$ must be done by the definition and it envolves considerations of too many action sequences. We expect to reduce their number, by considering only those that are necessary for the proof. We call these sequences **relevant action sequences**, and we are considering two methods to construct them: (i) choosing most general facts for input; (ii) finding necessary combinations of facts as input, in order to give output. Other proof techniques are also under development, especially to deal with recursion in the execution trees of CONESP.

Acknoledgements

The authors gratefully acknowledge G. Reggio for her comments on earlier drafts.

References

[1] V. Ambriola, P. Ciancarini, and C. Montangero. Software process enactment in Oikos. In R. N. Taylor, editor, *Proc. of ACM SIGSOFT '90, ACM Soft. Eng. Notes 15(6)*, Dec. 1990.

[2] E. Astesiano and G. Reggio. Direct semantics of concurrent languages in the SMoLCS approach. *IBM Journal of Research and Development*, 31(5), Sep 1987.

[3] E. Astesiano and G. Reggio. SMoLCS-Driven concurrent calculi. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science (249): Proc. of TAPSOFT'87*, Springer-Verlag, 1987.

[4] A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99 – 123, Jan. 1991.

[5] M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18, 1982.

[6] A. Bucci, P. Ciancarini, and C. Montangero. A distributed logic language based on multiple tuple spaces. In *Proc. of Logic Programming Conference*, Tokyo, July 1991 (to appear in LNCS).

[7] X. J. Chen. *A Formalism Towards Software Process Modelling.* PhD thesis, Scuola Normale Superiore, Pisa, Italy (in preparation).

[8] W. Deiters and V. Gruhn. Managing software processes in the environment Melmac. In R. N. Taylor, editor, *Proc. of ACM SIGSOFT '90, ACM Soft. Eng. Notes 15(6)*, Dec. 1990.

[9] G. E. Kaiser, P. H. Feller, and S. S. Popovich. Intelligent assistence for software development and maintenance. In *IEEE Software*, 1988.

[10] R. Milner. A calculus for communicating systems. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science (92)*, Springer-Verlag, 1980.

[11] N. H. Minsky and D. Rozenshtein. Configuration management by consensus: an application of Law-Governed systems. In R. N. Taylor, editor, *Proc. of ACM SIGSOFT '90, ACM Soft. Eng. Notes 15(6)*, Dec. 1990.

[12] G. Plotkin. *A Structural Approach to Operational Semantics.* Technical Report, Computer Science Deot. Aarhus Univ. Denmark, 1981. DAIMI-FN-19.

[13] G. Reggio. *Una Metodologia per la Specifica di Sistemi e Linguaggi Concurrenti.* PhD thesis, Dept. of Maths, Univ. of Genova, Italy, 1986.

[14] R. N. Taylor. Arcadia: a software development environment research project. In P. Henderson, editor, *Proc. of ACM SIGSOFT '88, ACM Soft. Eng. Notes 13(5)*, Nov. 1988.