Università di Pisa
Dipartimento di Informatica

Technical Report : TR-96-18

Applying Refinement Calculi to Software Process Modelling

C. Montangero, L. Semini

May 22, 1996

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

Applying Refinement Calculi to Software Process Modelling

Carlo Montangero Laura Semini

Dipartimento di Informatica, Università di Pisa
corso Italia 40, 56125 Pisa, Italy
tel. +39 (50) 887111, e-mail: {monta, semini}@di.unipi.it

Abstract

A refinement calculus provides a number of advantages to program development, besides correctness, like clarification of the goals of the program and effective documentation of the design choices. In this paper, we provide evidence that the same advantages can be obtained also in the case of those special programs that are known as enactable process models.

The evidence is put forward by means of an example, a small Concurrent Engineering problem inspired to the IWSP7 problem. We assume that the enactment is done by rules in tuple spaces, and use a refinement calculus based on a temporal logic that builds on that of Unity.

Finally, we show how the approach leads to seamless integration with existing process engines (the Oikos one in our case).

1 Introduction

The purpose of this work is to provide evidence that the stepwise development of Software Process Models, as advocated in Oikos [21], can be effectively supported by a refinement calculus; and that the approach results in seamless integration with existing Process Centered SDEs.

It follows from the need to communicate SP models to their users, like performers, customers and quality assurance people, that the approach presented here is relevant for the SP community. Indeed, it is well known, from the application of refinement calculi to program development (see e.g. [23]) that, besides supporting correctness, this approach leads to very well documented designs, where the rationale of each choice is clearly captured, and where the goals of the program are also clearly stated (forcedly, since the method needs an abstract specification to start with!). We aim to provide evidence that the same applies to those special programs that run in Process Centered SDEs, namely enactable process models.

The evidence will be put forward by the means of an example, that is inspired by what is known in the software process folklore as the IWSP7 problem. It is essentially an archetipal Concurrent Engineering [25] exercise: a number of developers is responsible for the concurrent development of a bunch of interconnected modules. Each developer is involved in several modules, and has a few requirements to fulfill per each of them. The purpose of the process is to fulfill all requirements on all modules. The rationale of this development approach, in which some work always has to be undone or adapted, since developers has to rely on assumptions on the work of others, or on modules that may be changed by others, is outside the scope of this paper.

The refinement calculus we will use is the one designed in the PhD thesis of the youngest author [26], which applies when the underlying paradigm is a multiple tuple spaces [11, 6].

The choice of the paradigm is not justified here: arguments in favor can be found both in the literature on the Software Process [5] and, more recently, on that on Coordination [8]. Suffice here to say that this choice introduces modularity, concurrency and distribution. Moreover, it will allow us to show, albeit sketchedly, the integration with the Oikos process engine [2].

The structure of the paper is the following. The next subsection puts our work in perspective with respect to software process modelling. Section 2 introduces the necessary technicalities (specification language, refinement and modularity concepts) to understand the examples. The first part of the example in Section 3 shows how to derive a rule from some of the goals in an abstract specification. Appendix A provides the flavor of the proofs that may be involved. Section 4 introduces an example of support to the interaction with a process engine, the Oikos Standard Desk, and shows how other goals in the abstract specification can be refined to match this support.

1.1 Related Works

In the Software Process literature, you find a moderate interest for process refinement, that follows at least two approaches, that we will call *explicit* and *implicit*.

The implicit approach can be loosely characterized as the attempt to exploit Artificial Intelligence techniques in software process modelling; it is exemplified by Articulator [20], Grapple [14], EPOS [9], and PEACE [3]. The emphasis is on the problem solving capabilities of the process engine, that enables the enactment of the process by refining an overall goal to simpler ones. Not always these approaches succeed in providing simple process models, nor it is always easy to assess them. Also modularity, concurrency and distribution issues may be hidden.

The explicit approach includes work that is sometimes similar to the previous one, e.g. because it also uses *rules*: the difference is here that the emphasis is descriptive. Examples are Merlin [16], Marvel, and its successors [5]. The original intent is that of Prolog, i.e. to provide a declarative description of the process, but then the needs of enactment get in the way, and the process model always gets an operational flavor.

The explicit approach also includes procedural refinement, with different degrees of declarativity. For instance HFSP [19], is based on a refinement approach, namely functional decomposition, and ends up with the usual operational flavour.

Essentially, all the previous attempts share the problems due to a single view of the building blocks, let them be goals, rules, function calls. To put it in another way, all of them exploit a single language, where the building blocks have at most a syntactic presentation, but their semantic definition is given either informally or in the same language: as a consequence it is difficult to keep the necessary distinction between the global goals of a model fragment, and the local constraints, as advocated for instance in [18]. We support this attitude, that we consider one of the ways to reach a good understandability of process models. Indeed, this is one of the goals of our work in SPT, as already mentioned in the introduction. In a programming view based on refinements, this decoupling is obtained by using two seamlessly integrated languages, one to describe semantically the interface of a module/piece, and the other that represents the target of the refinement, i.e. it expresses the implementation.

A step in the direction of separating specification from implementation is visible in Arcadia [17], since its language APPL/A inherits the package mechanism from Ada: however, the limitation of not having a language to provide a semantic description persists. In Oikos [21], some steps were taken, but relying on a very weak mechanism, namely path expressions: some of the semantics of a piece of a model, called *angel* in Oikos, is captured by the description of the allowed traces of the actions that the piece can take.

In this work we propose to exploit a full fledged first order temporal logic to provide the semantic characterization of the process fragments and tuple space rules as executable code.

We can so rely on the work in [26], to have a calculus that allows the refinement down to executable code, and we here specialize it to Software Process Modelling.

2 Background

We adopt the notion of *heterogeneous* system singled out in [26], to represent software processes. A system Σ consists of properties, rules, and an initial state. Methodologically, we start from a set of properties that are either goals or constraints[1] that the process must fulfill, and we arrive to a set of rules that enact it correctly, by calculated refinements. A refinement step is guided by the goals (we split a goal in smaller ones or we introduce rules "doing something good" and realizing a goal), and must satisfy the constraints (the introduced properties or rules "don't have to do anything bad"). The refinement laws for heterogeneous systems allow to cancel the goals that are fulfilled by the introduced rules, possibly substituting them with suitable constraints. When all the goals are fulfilled, the heterogeneous system consists of rules and constraints only. Again, the conditions we impose on refinement steps ensure that the introduced rules satisfy the constraints.

We classify system constraints as *global* or *local*. Global constraints refer to the broad evolution of the system, while local constraints state the conditions that depend upon the particular state encoding chosen, and are *local* to it.

An heterogeneous system Σ takes the form:

$$\Sigma : [\quad \begin{array}{ll} goal: & L \\ constraints: & S \\ local\ constraints: & LS \\ rules: & R \\ init: & Is \quad] \end{array}$$

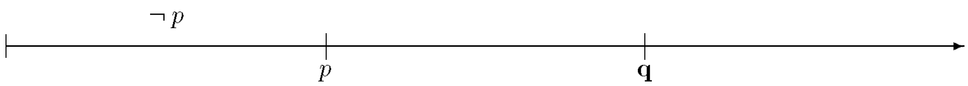
with L set of goals; S and LS sets of constraints; R set of rules; Is initial state. Goals and constraints are expressed in the specification language presented in the next section, rules are in a tuple space language [26, 8]. Each of these components may be empty[2], in which case we will simply not list them.

2.1 Specification Language

We adopt the specification language introduced in [26], which is based on first order temporal logic. The temporal operators are partly inherited from Unity [7]: $init$, $\mapsto$ (*leads-to*), $stable$, $unless$, and partly introduced ex-novo: $needs$, $after$, $\hookrightarrow$ (*causes*). An informal, graphical definition is in Table 1, and the formal definition is in Table 2.

The new operators are a sort of *past* operators and are introduced to enhance the expressive power of temporal logic when describing reactive systems as the ones we are taking into consideration, leading to more readable and concise specifications. In particular, they allow us to specify conditions related to the appearance (instead of presence) of a tuple in the tuple space. In other words they allow us to directly relate state changes to state configurations or to further reactions.

We need to discuss the case of $\mapsto$ versus $\hookrightarrow$. It can be shown that $p \mapsto q$ is stronger than $p \hookrightarrow q$, since every model of $p \mapsto q$ is also a model of $p \hookrightarrow q$. The goal $p \hookrightarrow q$ requires a state satisfying the conclusion (q) to follow the states in which the premise (p) **becomes** true. For instance, the following computation[3] is a model for $p \hookrightarrow q$:



[1] In temporal logic terminology, they correspond to liveness and safety properties, respectively.
[2] Actually, in the first refinement step $R = \emptyset$, in the last one $L = S = SL = \emptyset$.
[3] We are using the same conventions introduced in the caption of Table 1.

p(X) unless q(X) once true, p(X) holds until q(X) becomes true	
stable p(X) once true, p(X) stays true	
p(X) $\mapsto$ q(X) q(X) eventually true after p(X) was true	
p(X) needs q(X) q(X) must be true when p(X) becomes true	
p(X) after q(X) q(X) must be true before p(X) becomes true	
p(X) $\hookrightarrow$ q(X) q(X) eventually true after p(X) became true	

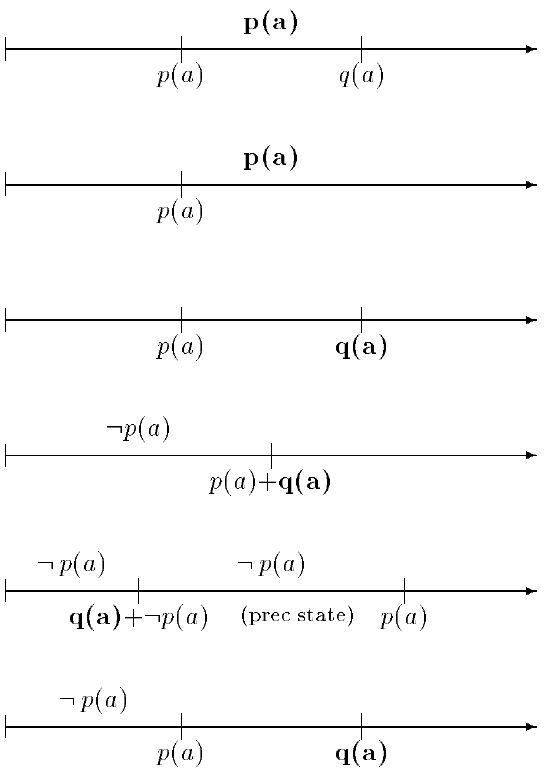


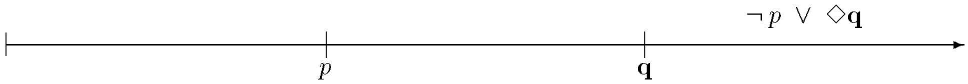
Table 1: Informal definition of temporal operators: predicates over the line refer to the time interval; predicates under the line refer to the time instant; predicates in boldface are those "caused" by the other conditions.

$$
\left.\begin{array}{lcl}
p \;\; unless \;\; q & \stackrel{def}{=} & \Box\,[\,(p \wedge \neg q) \rightarrow \bigcirc(p \vee q)\,] \\
stable \;\; p & \stackrel{def}{=} & p \;\; unless \;\; false \\
p \;\mapsto\; q & \stackrel{def}{=} & \Box\,[\,p \;\; \rightarrow \;\; (\Diamond\, q)\,]
\end{array}\right\}\, Unity
$$

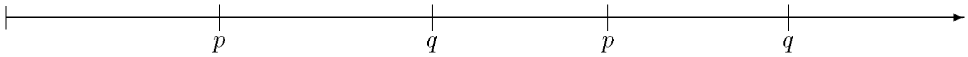
$$
\left.\begin{array}{lcl}
p \;\; needs \;\; q & \stackrel{def}{=} & init\;(p \rightarrow q) \wedge \Box\,[\,p \rightarrow (\hat{\bigcirc} p \vee q)\,] \\
p \;\; after \;\; q & \stackrel{def}{=} & \Box\,[\,p \rightarrow \hat{\bigcirc}(p \vee q)\,] \\
p \;\hookrightarrow\; q & \stackrel{def}{=} & p \;\; needs \;\; (\Diamond\, q)
\end{array}\right\}\, new
$$

Table 2: Formal definition of temporal operators: recall that $\Box$ means *always*; $\Diamond$ *eventually*; $\bigcirc$ and $\hat{\bigcirc}$ *next* and *previous* state.

i.e. each state in which p becomes true is followed by a state in which q is true. There are no other constraints: for instance, p can stay true even after the state in which q is true, without requiring q to stay or become true again. Instead, a computation which is a model for $p \mapsto q$ has to satisfy more constraints:



A state in which p is true is followed by one in which q is true: to let q undetermined after it appeared, we have to negate p. If p continues to be true after q became true, then further states with q true are required:



In other words, with $\mapsto$ we cannot falsify the conclusions without also falsifying the premises. This is why the use of $\hookrightarrow$ is very useful in the earliest stages of a refinement, when it is more convenient to say something like: "if p appears, then ..." deferring the choices related to p staying true or not.

For instance, let us consider goal DG of section 3.2:

$$job(X, R) + e\_current(\_, L) \quad \hookrightarrow \quad ready(X, \_, R, L') + L \leq L'$$

Its premise says that developer X has been assigned the job to fulfill requirement R, and that the current version of the procuct already satisfies the requirements in list L[4]. The consequence states that X has produced a work, that fulfills requirement R (besides others). The goal itself states that, following the state in which the premise **becomes** true, the process will reach a state, in which the consequence holds: nothing is implied on the future evolution.

Note that, in tuple spaces, the natural way to falsify a condition is to consume a tuple: for instance, once the version referred to in the fact ready as been dealt with, it is natural to consume it. With DG above, the refiner is free to follow this line. If, on the contrary, we substitute $\hookrightarrow$ with $\mapsto$ in DG, we are forcing the refiner to consume some piece of the premises also, if he wants to consume the consequence.

As a further example of the specification language, let us consider $p(X)$ *needs* $q(X)$. Such a constraint requires that, if $p(X)$ is true in a state s and false in the preceding state, i.e. $p(X)$ *appears* in s, then $q(X)$ must be true in s as well. We want X in $p(X)$ to bind the X occurring in $q(X)$.

We also use an operator $+$, a not-idempotent conjunction, which resembles the $\otimes$ (*times*) connective of linear logic [13]. Indeed, the states of a computation are multisets, in tuple space languages. We want to reflect the not-idempotent multiset union in the logic framework: with the operator $+$ we describe the tuple space $\{a, a\}$ simply by $a + a$.

2.2 Refinement

We base our calculus on assertions of the kind $F' \rhd F$ (F' refines F), and $R \rhd F$ (R refines F), where F, F' are formulae and R a rule. In the first case the relation $\rhd$ is the logical implication. In the second case, it is shown in [26] that it is enough to consider only the special cases $R \;\rhd\; p \, unless \, q$ and $R \;\rhd\; p \mapsto q$.

Intuitively, $R \;\rhd\; p \;\mapsto\; q$ allows to introduce rule R to fulfill goal $p \;\mapsto\; q$. Therefore, this refinement requires that R can be applied in a state in which $p \wedge \neg\, q$ holds and that its execution leads to a state in which q holds. Complementarily, $R \;\rhd\; p \, unless \, q$ requires that whenever R can be applied in a state in which $p \wedge \neg\, q$ holds, then it leads to a state in which $p \vee q$ holds, i.e. its execution lets the constraint invariantly satisfied.

[4] A side constraint ensures that R is not in L.

Formally, the refinement relation for rules is defined exploiting an axiomatic semantics [26]: we look at the rules as predicate transformers. For technical reasons we have to define two kinds of preconditions that can be calculated from a rule and a postcondition. We distinguish between the weakest precondition (wp) that ensures that a state satisfying the postcondition is reached even with a *demonic* choice of the atoms in the tuple space, and the enabling precondition (ep) that ensures that the postcondition is reached on the basis of an *angelic* choice of the atoms. Then we have:

$$\begin{array}{l} R \rhd p\ unless\ q \ \Leftarrow\ (p \wedge \neg q) \rightarrow (wp(R, p \vee q) \vee \neg wp(R, true)) \\ R \rhd p \mapsto q \ \Leftarrow\ (p \wedge \neg q) \rightarrow ep(R, p) \end{array}$$

and we can verify the correctness of the refinement of a goal to a rule.

2.3 Modularity

We impose some restrictions on heterogeneous systems, to link its components: we require the rules not to violate the constraints nor to cancel the premises of a goal, and the initial state to satisfy the constraints. Formally, let Σ be an heterogeneous system, we want, for any rule R_i in the system and for every property of the kind $p \mapsto q$, $p \hookrightarrow q$ in L

$$R_i \ \rhd\ S \cup LS \qquad\qquad R_i \ \rhd\ p\ unless\ q$$

These restrictions make the approach in [26] different from other approaches to *modular refinement*. For instance, in the *rely-guarantee* approach, Jones makes a clear distinction between properties that the local state has to *guarantee* and properties on which we can *rely*, without coupling them in any way [15]. This generates some problems when trying to compose different systems. In our view, a system integrates, in a unique description, properties that are refined locally, and properties that will be refined by the environment, and the links between these parts are formally defined. The problems related to composition in this framework are well described in [1].

3 Example

The problem we address deals with cooperation and coordination issues in software development. The scenario is the following. A team of engineers works in a project: the product is divided in modules, which are however interdependent, e.g., because of *use* relations. A team manager assigns the tasks, i.e. which requirements must be fulfilled by who on which modules. We will assume that each module can be assigned only to a single developer[5], who however can work on several modules. The job assignment is outside the scope of the exercise. We assume that each developer can make reasonable assumptions on the outcome of the work of the others, e.g. on the base of the design documents, and that he will adapt his solutions to the evolution of the project, to get to a consistent set of modules.

3.1 Abstract specification

In our solution we envisage to arrive at a consistent product via a sequence of consistent partial versions of the project, that incorporate the new modules produced by the developers as soon as they become available. We assume a FIFO discipline, and non-deterministic choice in case of concurrent deliveries. Finally, to try to limit potentially useless work, we notify the developers as soon as a new version of the project is available.

We do not consider explicitly the possibility of backtracking: once a module as been incorporated into the current version there is no provision to go back to an earlier version.

[5] The consequence of dropping this requirement would show only late in the refinement, where one should consider the available tools, to choose among mutual exclusion or sharing.

global goal (**GG**): $job(\_, R) \;\mapsto\; e\_current(\_, L) \;+\; R \in L$

global correctness (**GC**): $e\_current(V+1, [R|L]) after(e\_current(V, L) + job(\_, R))$

requirements boundedness (**RB**): $true \mapsto stable(\neg job(\_, R))$

Table 3: Abstract Specification. Predicate $\in$ is defined by: $R \in [R|L]$. $R \in [X|L] :- R \in L$

However, since we do not require that job assignments are statically defined, but only that they are bounded, backtracking can be simulated by a new bunch of assignments, under the responsibility of the project manager.

The environment state is encoded by the following atoms:

$job(X, R) :-$	Developer X has been assigned requirement R, and the corresponding work is still in progress.
$e\_current(V, L) :-$	Requirements in list L have been fulfilled by the current version V of the project[6].

The abstract specification of environment ISPW-7 consists only of goals and constraints:

$$ISPW7 : [\;\; goal : \;\; \{\text{ GG }\} \;\; constraints : \;\; \{\text{ GC, RB }\} \;\;]$$

with GG, GC, and RB defined in Table 3. The global goal (**GG**) says that if a job with requirement R is assigned to a developer, then in a future point in time, there will be a version of the project fulfilling R. The constraint global correctness (**GC**) requires the current version to be updated only to fulfill an assigned requirement. Requirements boundedness (**RB**) says that, at some future point in time, the assignment of new jobs will stop. We do not constrain the initial state. Notably, we do not require the list of fulfilled requirements to be empty in the initial version of a project. This is done on purpose. We can model in this way a re-assignment of tasks of a set of modules that are a work in progress.

3.2 Refinement step

We want to split global goal (GG) in smaller goals. To this purpose, we distinguish the responsibilities of the team manager from those of the developers. The developers are modelled as *angels*, as introduced in [21], producing some useful response even without an explicit request. The team manager updates the current version. We encode the state by the atoms listed in the previous section, plus:

$ready(X, W, R, L) \;:-\;$ the work W done by X satisfies both R and the requirements in list L.

The goal of a developer is to produce a fact *ready* when he has fulfilled a requirement. The produced work W has be be consistent with respect to the most recent version of the document he knows (this is specified by list L). It is not ensured that this is actually the current version of the document: in the meanwhile somebody else could have produced a new version. In this case the work of X cannot be accepted, and X will "know" that a new version is available and continue to work. On the other side, the goal of the team manager is to maintain the current version of each document and check whether the work done by a

[6] e stands for environment, to distinguish this condition from the similar one introduced in section 4 for the developer.

developer goal **(DG)**:

$$job(X, R) + e\_current(\_, L) \hookrightarrow ready(X, \_, R, L') + L \leq L'$$

management goal **(MG)**:

$$e\_current(V, L) + ready(\_, \_, \_, L) \mapsto e\_current(V+1, [\_|L])$$

developer correctness **(DC)**:

$$ready(X, \_, R, L') \ \ needs \ \ job(X, R) + e\_current(\_, L) + L \leq L'$$

management correctness **(MC)**:

$$e\_current(V+1, [R|L]) \ \ after \ \ ready(X, \_, R, L) + e\_current(V, L)$$

Table 4: Global Evolution. Predicate $\leq$ is defined by: $L \leq L$. $L \leq [X|L'] :- L \leq L'$

developer is consistent with it or not. Only in the first case the work is accepted and the current version is updated accordingly.

We introduce the notion of *useful ready*: a $ready(\_, \_, \_, \_)$ is useful if it fulfills the same list L of requirements that are satisfied by the current version of the project (as defined by $e\_current$). Note that two or more *useful* ready from different developers may be present in the same state. In this case only one of them is selected to update the current version of the project. The others become immediately not useful and will not be used. The involved developer will angelically continue to work to produce a newer work to fulfill the larger list of requirements.

At this stage the system is modelled by:

$$\begin{array}{llll} ISPW7: [& goal: & \{ \text{DG, MG} \} & \\ & constraints: & \{ \text{DC, MC, RB} \} & \\ & local\ constraints: & lc &] \end{array}$$

with DG, MG, and DC defined in Table 4, RB defined in Table 3, and set lc defined in Table 5, and discussed below.

Goals. Developer goal (**DG**) ensures that, if a version of the project, fulfilling requirements in list L, is available in a state in which X is still working ($job(X, R)$), then X will produce a new work, fulfilling R and consistent with L, in a future point in time. Actually, the version produced by X can can refer to a more recent version of the project, this is why we say that L' has to include (and not to be equal to) L. Management goal (**MG**), ensures that, with a *useful* ready, the current version is updated. We cannot say the stronger

$$e\_current(V, L) + ready(\_, W, R, L) \mapsto e\_current(V+1, [R|L])$$

since we would require, this way, a fact $e\_current(\_, [R|L])$ to follow every $ready(\_, W, R, L)$ which is *useful*. On the contrary, when two or more *useful* ready from different developers are present in the same state, only one is chosen to update the current version.

In the appendix we prove that DG and MG refine GG.

Global constraints. The first constraint, developer correctness (**DC**) ensures that a developer delivers a new work only if the corresponding job is still open, and that each delivery can only try to fulfill requirement R, matching at the same time as many requirements already met by other developers as he knows of. In other words, the fulfilled requirements cannot exceed the list of requirements fulfilled by the current version of the document. Management correctness (**MC**), guarantees that the current version of the project can be updated only in correspondence to a useful ready.

consistency:	$\Box\ (\ job(\_,\_)\ \rightarrow\ e\_current(\_,\_)\ )$
no duplication of work:	$\Box\ \neg\ (\ job(\_,R)\ +\ job(\_,R))$
no completed jobs:	$\Box[(e\_current(\_,L)\ +\ job(X,R))\ \rightarrow\ R\notin L]$
fulfillment:	$job(X,R)\ unless\ e\_current(\_,[R\|L])$
persistency:	$e\_current(V,L)\ unless\ e\_current(V+1,[\_\|L])$
unique current version:	$\Box\ \neg\ (e\_current(\_,\_)\ +\ e\_current(\_,\_))$

Table 5: Local Constraints

Local constraints. The first local constraint, "consistency", says that it is not possible to have a job assigned, without the existence of a current version of the project. "No duplication of work" says that a requirement can be assigned only to one developer. With "no completed jobs" we require that, when a work with requirement R is completed, i.e. $e\_current(\_,L)\ +\ R\in L$, then the corresponding job is canceled: there are no jobs for completed requirements. Property "fulfillment" requires that a job is not cancelled unless it has been completed. With "persistency" we ask that the list of requirements satisfied by the current version of the project does not decrease. Finally, "unique current version" imposes the fact keeping the current version of the project to be unique.

3.3 Heterogeneous step

We make an heterogeneous step in our refinement. According to the method proposed in section 2, we introduce a rule fulfilling a goal and guaranteeing safety. We actually refine management goal (**MG**) with management rule (**mr**), which is derived as follows. We start from a rule schema having premises and conclusions of goal **MG** as **read-consume** and **tell** parts, respectively:

$$\begin{array}{ll} mr: & \textbf{read or cons}\ ready(\_,\_,\_,L) \\ & \textbf{read or cons}\ e\_current(V,L) \\ & | \\ & \textbf{tell}\ e\_current(V+1,[\_|L]) \end{array}$$

and then we fill it to satisfy the constraints given by the system[7] and the syntactic constrains requiring the variables appearing in the **tell** part of a rule to be bound. We are thus led to:

$$\begin{array}{ll} mr: & \textbf{read}\ ready(\_,\_,R,L) \\ & \textbf{cons}\ e\_current(V,L) \\ & \textbf{cons}\ job(\_,R) \\ & | \\ & \textbf{tell}\ e\_current(V+1,[R|L]) \end{array}$$

The manager is thus completely automated, and ISPW-7 is now modelled by the system:

$$\begin{array}{llll} ISPW7: [& goal: & \{\ \mathrm{DG}\ \} & \\ & constraints: & \{\ \mathrm{DC,\ MC,\ RB}\ \} & \\ & local\ constraints: & lc\cup\{safety(MG)\} & \\ & rules: & \{\ \mathrm{mr}\ \} &] \end{array}$$

where $safety(MG) = [e\_current(V,L)\ +\ ready(\_,\_,\_,L)]\ unless\ e\_current(V+1,[\_|L])$. The formal refinement can be performed as sketched in section 2.2.

[7] Management correctness, unique current version, fulfillment, and no completed jobs are the constraints direclty involved.

4 An example of integration with a PCSDE: the Oikos Standard Desk

We show here how we can integrate $ISPW7$ with the Oikos Software Development Environment. Namely, we want to refine developer goal (**DG**), that, remember, says that each requirement will eventually be implemented by the developer in charge. We want to allow a developer to keep the pace with the current versions of the modules it's working on, at its best. We introduce the following intermediate goal: when a new version of project becomes available, we want to attract the actor's attention, so that it can change its attitude, and *integrate* its own changes into the newer version, rather than continue to work on the obsolete one.

We will also focus on the following scenario: an engineer playing $dev(3)$ is working to fulfill $requir(6)$, and is aware of version 2 of the project that already fulfills a couple of requirements. In other words,

$$job(dev(3), requir(6))$$
$$e\_current(2, [requir(3), requir(5)])$$

hold in $ISPW7$. We want to reflect on $dev(3)$'s desk each new version of the project, whenever the manager accepts a new *useful ready* leading, for instance, to

$$e\_current(3, [requir(1), requir(3), requir(5)])$$

4.1 The Oikos Standard Desk

In Oikos, the people performing a process are called *actors*, and they play *roles* through a graphic interface (the so-called *Role-Pad*) which allows the actors to work on their *desks*, that hold the state of the iteraction and perform the actual access to tools and documents. The *standard desk* [22] summarises the experience in modelling cooperative processes in Oikos, providing a general flexible model for the specification of user support, guide and control in Oikos. The standard desk features: a declarative form of definition and control of the roles; an integrated access to the *Oikos Standard Services* [2]; the automatic control of the human-machine interface, the *Role-Pad*; some guidelines to the process modeller. The resulting style of interaction has been demonstrated at IWSP9 [24], and is similar to other approaches like the *workspaces* of Merlin [16], and the *agendas* of SPADE [4].

The standard desk allows the local customization, for each desk of the process, of the general model provided by the *Oikos Standard Services*: tools, documents and actions (like the exclusive access to a document) of each role are described synthetically an declaratively. An interpreter on the desk takes care of all the necessary interactions with the services, for each client (role).

We provide here a sketch of how to specify the standard desk so that it can be assumed as a target for the refinement of goal DG. At any given moment a role on the standard desk is in one of finite number of *interaction states*; each interaction state defines the actions the actor playing the role is allowed to perform on the product, i.e. which documents it can access, which tools it can invoke, and which communications it is allowed. The desk is reactive to the role's actions, i.e. these actions can cause state transitions. Besides, each document on the standard desk has an *application state*, that characterizes the actions that can be performed on it.

For instance, in the scenario above, $dev(3)$ will see on its *Role-Pad* icons representing both the modules it's working on and the requirements it's trying to fulfill. This means that

$$status(dev(3), working)$$
$$document\_status(dev(3), module(5), .., in\_use, read\_write)$$
$$document\_status(dev(3), requir(6), .., in\_use, read\_only)$$

standard desk goal **(SDG)**:

$$change\_state(X, S_1, S_2) + status(X, S_1) \quad \mapsto \quad status(X, S_2)$$

standard desk schema **(SDS)**:

$$send\_goad\_warning(X, FACT, Msg) \quad \hookrightarrow \quad FACT$$

Table 6: A fragment of the Standard Desk Specification

will hold. The dots list the actions which are allowed on the document: we need not dwell into these details here.

The standard desk changes state by obeying explicit commands, like the appearance of

$$change\_state(dev(3), working, integrating)$$

In other words, it satisfies goals like **SDG** of Table 6 so that we can freely use these properties as axioms in our refinement. Another property of the standard desk, which is relevant to our purposes here, is that it will react to requests like

$$\begin{array}{ll} send\_goad\_warning\,(& dev(3),\ change\_state(dev(3), working, integrating), \\ & \text{“}warning:\ newer\ version\ available\text{”} \qquad) \end{array}$$

by popping up an icon on dev(3)'s Role-Pad, showing the message string in it, and generate the fact

$$change\_state(dev(3), working, integrating)$$

once the actor playing the role clicks on the icon, to acknowledge the message. In other words, the standard desk gives us axiom schema **SDS** of Table 6, per any predicate FACT.

4.2 Refining Developer Goal

We need to introduce information about the current version of the project a role is working on[8], e.g.

$d\_current(dev(X), V) :-$ Developer X is working on version V of the project.

We can finally formulate our goal, that links $ISPW7$ and the desk, in the process of refining goal DG:

$$\begin{array}{ll} IG: & job(X, \_) + e\_current(V', \_) \\ & \quad + status(X, working) + d\_current(X, V) \ \wedge\ V' > V \\ & \hookrightarrow \\ & change\_state(X, working, integrating) \end{array}$$

The interesting point is that we likely want the user to decide whether to stop immediately or to clean up its current version before moving to the integration state. Then, we can further refine the goal introducing one that exploits the schema above and calls in the actor to produce the state transition command:

[8]The standard desk has no such concept: the version information is only used explicitely, when invoking the tools. Indeed, in the “integrating” state the role will work on two different versions of the same module.

$$
\begin{array}{ll}
IG' : & job(X,R) + e\_current(V',L) \\
& \quad + status(X, working) + d\_current(X,V) \ \wedge \ V' > V \\
& \hookrightarrow \\
& \quad send\_goad\_warning(\quad X, change\_state(X, working, integrating), \\
& \qquad\qquad\qquad\qquad\quad \text{“}warning: \ newer\ version\ available\text{”} \qquad)
\end{array}
$$

5 Conclusions

To conclude, we show how our approach may produce the advantages claimed in the introduction. When, as we did in section 4.2, we use the standard desk schema SDS to ensure that a given condition is established, we assume that the actor *will* acknowledge the message. We can push the exercise further, and model the expected behaviour of the actor, rephrasing the schema as

$$
\begin{array}{ll}
& posted(X, Msg, Window\_id) \\
& \hookrightarrow \\
& clicked(Window\_id) + \neg\, posted(X, Msg, Window\_id) \\
\Rightarrow & \\
& send\_goad\_warning(X, FACT, Msg) \\
& \hookrightarrow \\
& FACT
\end{array}
$$

For documentation purposes, we can then collect the application conditions that have been actually used in the refinement, as a set of goals for the performers:

$$
\begin{array}{l}
posted(\_, \text{“}warning: \ newer\ version\ available\text{”}, W) \\
\hookrightarrow \\
clicked(W)
\end{array}
$$

Moreover, some constraints may be derived just for documentation purposes: for instance, the meaning ot the warning messages can be systematically captured, like

$$
\begin{array}{l}
posted(X, \text{“}warning: \ newer\ version\ available\text{”}, \_) \\
needs \\
status(X, working) + d\_current(X,V) + e\_current(V', \_) \ \wedge \ V' > V
\end{array}
$$

which can be proved from the original goals and constraints.

Finally, the refinement calculus can support also the decomposition of a system in separate modules, that can then be enacted on a distributed architecture. The paradigm still exploits shared memory, with both *local* and *remote* writes and only *local* reads; we assume no global syncronization.

For instance, an alternative refinement of developer goal (DG), might introduce the environment $ISPW7$ and the desks $desk(\_)$ as separate modules, to be composed in parallel:

$$ISPW7 \ \| \ desk(X) \ \| \ \ldots \ \| \ desk(Y)$$

The new properties use predicate

$job\_at\_desk(V,R,L) :-$ notifies that the work to fulfill requirement R has also to be consistent with the current version of the project V already fulfilling requirements in list L.

to decribe the state of a desk. The properties listed in Table 7 refine goal DG of Table 4. Goal IS ensures that every time the *e_current* version of the project is updated to V, the developers which are still working are notified, albeit remotely. Developer goal′ DG′ ensures that, if X is notified to fulfill R and to be consistent with L, it will produce work W with integration stymulus (**IS**):

$$(e\_current(V,L)\ +\ job(X,R))\ \hookrightarrow\ job\_at\_desk(V,R,L)@X$$

developer goal$'$ (**DG$'$**):

$$job\_at\_desk(\_,R,L)\ \mapsto\ (ready(X,W,R,L')@ISPW7\ +\ L \leq L')$$

Table 7: Properties of the system $ISPW7 \parallel desk(\_)$. In the logical framework, the sentence $P@N$ means that P will eventually hold in X, i.e. that the remote write of P into N has been initiated.

such qualities, in a future point in time. As we did before, we do not force X to answer every notification: this is why L' has to include, not to be equal to, L, and we do not refer to a given version.

It is well known that the key point with refinements that cope with module decomposition is related to how to link local and remote conditions, i.e. how to deal with the @ operator, in our setting. The necessary refinement laws are presented in [26]: their application to Software Processes will be fully explored in a forthcoming paper.

References

[1] M. Abadi and L. Lamport. Composing Specifications. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Proc. Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 1–41. Springer-Verlag, Berlin, 1989. Invited Lecture.

[2] V. Ambriola, G.A. Cignoni, and C. Montangero. The Oikos Services for Oject Management in the Software Process. In B.C. Warboys, editor, *Proc. 3rd European Workshop on Softare Process Technology, EWSPT '94*, volume 772 of *Lecture Notes in Computer Science*, pages 2–14, Villard de Lans, France, 1994. Springer-Verlag.

[3] S. Arbaoui and F. Oquendo. PEACE: Goal-oriented Logic-Based Formalism for Process Modelling. pages 249–278. In [10].

[4] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: An Environment for Software Process Analysis, Design, and Enactment. pages 223–247. In [10].

[5] I. Ben-Shaul and G. Kaiser. *A paradigm for decentralized process modeling*. Engineering and computer science. Kluwer Academic Publishers, Boston, 1995.

[6] A. Bucci, P. Ciancarini, and C. Montangero. A Distributed Logic Language Based on Multiple Tuple Spaces. In *Proc. Logic Programming Conference*, Tokio, 1991.

[7] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading Mass., 1988.

[8] P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models, First International Conference, COORDINATION 96*, volume 1061 of *Lecture Notes in Computer Science*, Cesena, April 1996. Springer-Verlag.

[9] R. Conradi, M. Hagaseth, J. Larsen, M. N. Nguyen, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. EPOS: Object-oriented Cooperative Process Modelling. pages 33–70. In [10].

[10] A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Advanced Software Development. Research Study Press distributed by J. Wiley & sons, London, 1994.

[11] D. Gelernter. Multiple Tuple Spaces in Linda. In *Proc. Parle 89*, volume 365 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[12] C. Ghezzi, editor. *Proc 9-th Int. Workshop on the Software Process*, Airlie, Va, Oct 1994. IEEE Press.

[13] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

[14] K. Huff and V. Lesser. A Plan-based Intelligent Assistant that supports the Software Development Process. In P. Enderson, editor, *ACM SIGSOFT '88*, pages 97–106, Boston MA, 1988. ACM Press.

[15] C.B. Jones. Specification and Design of (Parallel) Programs. In R.E.A. Mason, editor, *Information Processing 83*, pages 321–332. Norh-Holland, 1983.

[16] G. Junkermann, B. Peuschel, W. Schaefer, and S. Wolf. MERLIN: Supporting Cooperation in Software Development through a Knowledge-Based Environment. pages 103–130. In [10].

[17] R. Kadia. Issues Encountered in Building a Flexible Software Development Environment. In H. Weber, editor, *5th ACM SIGSOFT*, pages 169–180, Tyson's Corner VA, 1992. ACM Press.

[18] G.E. Kaiser, S.S. Popovich, and I.Z. Ben-Shaul. A Bi-Level Language for Software Process Modeling. In *Proc. 15th International Conference on Software Engineering*, pages 132–143, Baltimore, Mariland, May 1993.

[19] T. Katayama. A Hierarchical and Functional Software Process Description and its Enaction. In *Proc. 11th Int. Conf. on Software Engineering*, pages 343–351. IEEE Computer Society Press, 1989.

[20] P. Mi and W. Scacchi. Modeling Articulation Work in Software Engineering Processes. In M. Dowson, editor, *First Int. Conf. on the Software Process*, pages 188–201, Redondo Beach, CA, 1991. IEEE Computer Society Press.

[21] C. Montangero and V. Ambriola. Oikos: Constructing Process-centered SDEs. pages 131–151. In [10].

[22] C. Montangero and S. Tirabassi. Desks and Roles in Oikos: Modeller's Manual. Technical Report 6/118, Progetto Finalizzato CNR "Sistemi Informatici e Calcolo Parallelo", 1994. (in italian).

[23] C.C. Morgan. *Programming from Specifications.* Prentice-Hall, 1990.

[24] L. Penedo. Report on the demo day. In [12].

[25] Y. V. Ramana Reddy, Kanakanahalli Srinivas, V. Jagannathan, and Raghu Karinthi. Computer Support for Concurrent Engineering. *IEEE Computer*, 26(1):12–15, 1993.

[26] L. Semini. *Refinement in Tuple Space Languages.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1996. TD 10-96. At www.di.unipi.it/˜semini.html.

A Refinement Proof

We prove here that management and developer goal (MG and DG) refine global goal (GG). We first prove that if a job with requirement R is assigned to a developer, then in a future point in time, either there will be a version of the project fulfilling R or the state in which new jobs cease to be assigned.

$job(\_, R)$

$\Rightarrow$ {consistency, L is existentially quantified}

$job(\_, R) + e\_current(\_, L)$

$\mapsto$ {requirements boundedness, persistency}

$stable(\neg job(\_, \_)) + e\_current(\_, L')$

$\Rightarrow$ {fulfillment}

$stable(\neg job(\_, \_)) +$
$(e\_current(\_, L') + R \in L \vee e\_current(\_, L') + job(\_, R))$

$\equiv$

$[e\_current(\_, L') + R \in L] \ \vee \ [stable(\neg job(\_, \_)) + e\_current(\_, L') + job(\_, R)]$

In the first case the goal is satisfied, otherwise we prove that, under the hypothesis $stable(\neg job(\_, \_))$, either we satisfy the goal or we strictly reduce the number of works in progress. Let $other(R, N) \equiv$

$$\underbrace{job(\_, \_) + \ldots + job(\_, \_)}_{N\ times} + \neg\,[\underbrace{job(\_, \_) + \ldots + job(\_, \_)}_{N+1\ times} + job(\_, R)]$$

i.e. N is the number of jobs different from $job(\_, R)$:

$e\_current(\_, L) + job(\_, R) + other(R, N)$

$\mapsto$ {developer goal, developer correctness}

$job(\_, R) + e\_current(\_, L') + ready(\_, \_, R, L'') + L \leq L'' \leq L'$

$\Rightarrow$

$job(\_, R) + e\_current(\_, L) + ready(\_, \_, R, L) \vee$
$job(\_, R) + e\_current(\_, L') + L < L'$

$\equiv$ {fulfillment, no competed jobs}

$job(\_, R) + e\_current(\_, L) + ready(\_, \_, R, L) \vee$
$job(\_, R) + e\_current(\_, L') + other(R, N') + N' < N$

The second case says that the number of works in progress has reduced. Let us consider the first case:

$job(\_, R) + e\_current(\_, L) + ready(\_, \_, R, L) + other(R, N)$

$\mapsto$ {management goal}

$e\_current(\_, [R'|L\})$

$\Rightarrow$ {fulfillment, no competed jobs }

$e\_current(\_, [R|L\}) \ \vee \ [other(R, N') + N' < N]$