Combining Hierarchical Specification with Hierarchical Implementation

Naijun Zhan

Lehrstuhl für Praktische Informatik II
Fakultät für Mathematik und Informatik
Mannheim Universität
D7,27, 68163 Mannheim, Deutschland
zhan@pi2.informatik.uni-mannheim.de

Abstract. Action refinement is a practical hierarchical method to ease the design of large reactive systems. Relating hierarchical specification to hierarchical implementation is an effective method to decrease the complexity of the verification of these systems. In our previous work [15], this issue has been investigated in the simple case of the refinement of an action by a finite process.
In this paper, on the one hand, we extend our previous results by considering the issue in general, i.e., refining an abstract action by an arbitrary process; on the other hand, we exploit different techniques such that our method is easier to be followed and applied in practice.

Keywords: Action refinement, modal logic, specification, verification, reactive system.

1 Introduction

Generally speaking, it is not easy, even impossible to capture a complex system at the beginning. *The hierarchical development method* is one of the practical and effective methods for designing large systems by specifying and implementing a system at different levels of abstraction. In process algebraic settings, *action refinement* [8] is such a kind of methods. We are here interested in the question how verification can be incorporated in the hierarchical development. In particular, we investigate how action refinement can be incorporated into a specification logic in such a way that it mimics the refinement in the process algebra. In the literature, some first attempts to solve this problem are given, for example in [10,13,14].

The main results obtained in [10,13,14] are as follows: Given an abstract specification ϕ in some logic, say the modal μ-calculus, and a model P of a complex system, and a refinement Q for a primitive a in P, where Q is a finite process, build $P[a \rightsquigarrow Q]$ and $\phi[a \rightsquigarrow Q]$ as the refinements of the model P and the specification ϕ respectively. [10] and [13,14] deal with $P[a \rightsquigarrow Q]$ in different way, but all define $\phi[a \rightsquigarrow Q]$ by replacing $\langle a \rangle$ and $[a]$ in ϕ by some formulae of the forms $\langle a_1 \rangle \langle a_2 \rangle \dots \langle a_n \rangle$ and $[a_1][a_2] \dots [a_n]$ respectively, where $a_1 a_2 \dots a_n$ is

V.A. Saraswat (Ed.): ASIAN 2003, LNCS 2896, pp. 110–124, 2003.

a run of Q. Then they prove that $P \models \phi$ iff $P[a \rightsquigarrow Q] \models \phi[a \rightsquigarrow Q]$ under some syntactical conditions.

In the above approaches, the refinements of the specification and the model are explicitly built on the structure of Q. This restricts the refinement step in two ways: firstly, there are some desired properties of the refined system that cannot be deduced in the setting of [10,13,14]. For example, let $P = a; b + a; c$, $\phi = \langle a \rangle$, $Q = a'; (c'; b'; d' + c'; b')$. It's obvious that $P \models \phi$ and $Q \models \langle a' \rangle [c'] \langle b' \rangle$. It is expected that $P[a \rightsquigarrow Q] \models \langle a' \rangle [c'] \langle b' \rangle$. But it cannot be derived using the approaches of [10,13,14]; secondly, the refinement step is restricted to one choice of Q for refining an action a, which appears both in the refined process and the refined specification explicitly.

In contrast to this, in [15] we proposed a general approach on how to construct a low-level specification by refining the higher-level specification. But as in [10,13,14], we also only considered the simple case to refine an abstract action by a finite process. The basic idea is to define a refinement mapping Ω which maps the high-level specification ϕ and the property ψ of the refinement Q of an abstract action a to a lower-level specification by substituting ψ for $\langle a \rangle$ and $[a]$ in ϕ. Since ψ can be any property that holds in Q, we can get the expected specification if ϕ and ψ are appropriate. For example, in the above example, we can get $\Omega(\phi, \langle a' \rangle [c'] \langle b' \rangle, a) = \langle a' \rangle [c'] \langle b' \rangle$ which is exactly what we expect. However, Q can only be any finite process, which implies that ψ is essentially equivalent to a formula without fixpoint operators.

But, in many applications, an action has to be refined by a process with potentially infinite behaviour. For example, in the programming, we can look the interface of a procedure as an abstract action and its body as its refinement. In an abstract level, we only need to use the interface instead of the procedure, but it is necessary to substitute the body for the interface when the procedure is considered in a lower level. In many cases, we need to implement a procedure with possibly infinite behaviour in order to meet the given requirements. For instance, in the example of a salesman [15] (It can be found in Section 4), if we know that the job of the salesman in London every day is repeatedly to meet some of his customers in his office or contact some of them by phone, the action "work" in the top-most specification should be refined by the above concrete procedure in the lower-level specification. However, it is obvious that such a job can not be done using our previous approach.

So, in order to have more applications, we extend our previous work by refining an abstract action by an arbitrary process in this paper. To this end, we adopt FLC as specification language.

FLC is due to Müller-Olm [16], and is an extension of the μ-calculus by introducing the chop operator ";". FLC is strictly more expressive than the μ-calculus because the former can define non-regular properties [16], whereas the latter can only express regular properties [7,9]. The model-checking of FLC was addressed in [11,12]. For technical reasons, here we augment FLC by introducing a special propositional constant $\surd$ to indicate if a process is terminated and re-interpret $[a]$ appropriately.

As discussed in [15], a sound refinement mapping should keep the type of properties to be refined, i.e. an existential property should be refined to an existential property and similarly for the other properties. In order to define a refinement mapping that can preserves the type of the property to be refined, as in [15], the property for the refinement Q will be partitioned into two subformulae: an *universal formula* and an *existential formula*. The former will be used to substitute for $[a]$ and the latter for $\langle a\rangle$ in ϕ. Such partition is justified by the result proved in [4] that every property can be represented as the conjunction of a safety property and a liveness property in branching models. Besides, we require that ψ is only relevant to full executions of Q. If so, a refinement mapping that keeps the type of the property to be refined can be defined like in [15]. Furthermore, we can prove the following theorem:

Theorem. (Refinement Theorem) *If some syntactical conditions hold,* $P \models \phi$ *and* $Q \models \psi; \surd$ *then* $P[a \rightsquigarrow Q] \models \Omega(\phi, \psi, a)$.

The above theorem supports 'a priori' verification in the following sense: In the development process we start with $P \models \phi$ and either refine P and obtain automatically a (relevant) formula that is satisfied by $P[a \rightsquigarrow Q]$; or, we refine ϕ using $\Omega(\phi, \psi, a)$ and obtain automatically a refined process $P[a \rightsquigarrow Q]$ that satisfies the refined specification. Of course such refinement steps may be iterated.

To achieve the intended result, we need to assume that action refinement for models is atomic. Our main aim in this work is to establish a correspondence between hierarchical implementation and hierarchical specification of a complex system. But if we allow that the refining process can be interleaved with others problems will arise. E.g. $(a \parallel_{\{\}} b)[a \rightsquigarrow a_1; a_2]$ means the parallel executions of a and b in which a is refined by $a_1; a_2$. It's obvious that $a \parallel_{\{\}} b$ satisfies $\langle a\rangle$, and $a_1; a_2$ satisfies $\langle a_1\rangle; (\langle a_2\rangle \wedge [b]; false)$ which means that $a_1; a_2$ first performs a_1, then a_2 but cannot perform b. We expect that $a \parallel_{\{\}} b$ meets $\langle a_1\rangle; (\langle a_2\rangle \wedge [b]; false)$ after refining a by $a_1; a_2$. This is not true in the case of non-atomic action refinement since b can be performed between the execution of a_1 and a_2. But it is valid if we assume that action refinement is atomic [5,6]. So, in the sequel, we discuss action refinement for models under the assumption of atomicity.

Besides, we will exploit different techniques such that all the results proved in this paper are represented in a simpler way and easier to be used in practice.

The remainder of this paper is organized as follows: A modeling language is defined in Section 2; Section 3 briefly reviews FLC; A refinement mapping for specifications is given in Section 4; The correspondence between the hierarchical specification and the hierarchical implementation of a complex system is shown in Section 5; Finally, a brief conclusion is provided in Section 6.

2 Modeling Language – A TCSP-like Process Algebra

2.1 Syntax

As in [14], we use a TCSP-like process algebra in combination with an action refinement operator as modeling language. Let Act be an infinite set of (atomic) actions, ranged over by $a, b, c, \ldots$, and A be a subset of Act. Let $\mathcal{X}$ be a set of

process variables, ranged over by $x, y, z, ...$ The language of processes, denoted by $\mathcal{P}$ and ranged over $P, Q, \ldots$, is generated by the following grammar:

Definition 1.

$$P \ ::= \ \delta \mid nil \mid a \mid x \mid P; Q \mid P + Q \mid P \parallel_A Q \mid rec\, x.P \mid P[a \rightsquigarrow Q]$$

where $a \in Act, x \in \mathcal{X}$, *and* $P, Q \in \mathcal{P}$.

An occurrence of a process variable $x \in \mathcal{X}$ is called *bound* in a process term P iff it does occur within a sub-term of the form $rec\, x.P'$, otherwise called *free*. A process expression P is called *closed* iff all occurrences of each variable occurring in it are bound, otherwise it is called *open*. We will use $fn(P)$ to stand for the variables that have some free occurrence in P, $bn(P)$ for the variables that have some bound occurrence in P. When we say a process P is *terminated*, it means that P does nothing except for terminating (see Definition 2). A variable $x \in \mathcal{X}$ is called *guarded* within a term P iff every occurrence of x is within a sub-term Q where Q lies in a subexpression $Q^*; Q$ such that Q^* is not terminated. A term P is called *guarded* iff all variables occurring in it are guarded. Sometimes, we abuse $Act(P)$ to stand for the set of actions which occur in P.

For technical reasons, as in [8], we require the following well-formedness conditions on $\mathcal{P}$:

- **(i)** None of operands of $+$ is a terminated process;
- **(ii)** All process terms are guarded;
- **(iii)** The refinement of an action can not be a terminated process. As discussed, e.g. in [17], refining an action by a terminated process is not only counter-intuitive but also technically difficult.

Intuitively, $P[a \rightsquigarrow Q]$ means that the system replaces the execution of an action a by the execution of the subsystem Q every time when the subsystem P performs a. This operator provides a mechanism to hierarchically design reactive systems. The other expressions of $\mathcal{P}$ can be conceived as usual ones. The formal interpretation of $\mathcal{P}$ will be provided in the next section.

2.2 Operational Semantics

Here we define an operational semantics for $\mathcal{P}$ employing transition systems. The meaning of the constructs of the language can be interpreted in the standard way except for the refinement operator. In order to guarantee the atomicity of the refinement, the basic idea is to define a transition system for the process that may be refined, then replace all transitions labelled with the action to be refined by the transition system for the refinement.

Similar to [8], the above idea can be implemented by introducing an auxiliary operator $*$ to indicate that a process prefixed with it is the remainder of some process, which has the highest precedence and must be performed completely. The state language, denoted by $\mathcal{P}^*$, ranged over by $s, \ldots$, is given by:

$$s \ ::= \ nil \mid \delta \mid a \mid x \mid *s \mid s; s \mid P + Q \mid s \parallel_A s \mid s[a \rightsquigarrow Q] \mid rec\, x.P$$

where $a \in Act, x \in \mathcal{X}, P, Q \in \mathcal{P}$.

According to the above definition, it is clear that $\mathcal{P}$ is a proper subset of $\mathcal{P}^*$, i.e. $\mathcal{P} \subset \mathcal{P}^*$.

In order to define the semantics of $\mathcal{P}^*$, we need the following definition.

Definition 2. *Let* $\surd$ *and* $\mathbf{ab}$ *be the minimal relations on* $\mathcal{P}^*$ *which satisfy the following rules, respectively:*

$\surd(nil)$	$\dfrac{\surd(s)}{\surd(*s)}$	$\dfrac{\surd(s_1) \wedge \surd(s_2)}{\surd(s_1 \parallel_A s_2)}$
	$\surd(rec\, x.s)$	$\surd(s_1; s_2)$
	$\surd(s[a \rightsquigarrow Q])$	*where* $Q \in \mathcal{P}$

Definition of $\surd$

$\dfrac{\surd(s)}{\mathbf{ab}(s)}$	$\mathbf{ab}(a)$	$\dfrac{\mathbf{ab}(s_1) \wedge \mathbf{ab}(s_2)}{\mathbf{ab}(s_1; s_2)}$	$\dfrac{\mathbf{ab}(s)}{\mathbf{ab}(rec\, x.s)}$
	$\mathbf{ab}(x)$	$\mathbf{ab}(s_1 + s_2)$	$\mathbf{ab}(s[a \rightsquigarrow Q])$
		$\mathbf{ab}(s_1 \parallel_A s_2)$	*where* $Q \in \mathcal{P}$

Definition of $\mathbf{ab}$

Note that in the above definition, $\surd(s)$ means that s is terminated, whereas $ab(s)$ means that s is either in $\mathcal{P}$, or terminated. A state s is called *abstract* if $\mathbf{ab}(s)$, otherwise, called *concrete*.

Besides complying with the three well-formedness conditions for $\mathcal{P}$, $\mathcal{P}^*$ also follows the below well-formedness condition:

(iv) At least one of the operands of $\parallel_A$ is abstract.

An operational semantics of $\mathcal{P}^*$ is given by the following transition rules:

Act $\quad a \xrightarrow{a} nil$

Nd $\quad \dfrac{P \xrightarrow{a} s}{P + Q \xrightarrow{a} s \text{ and } Q + P \xrightarrow{a} s}$

Seq-1 $\quad \dfrac{s_1 \xrightarrow{a} s_1'}{s_1; s_2 \xrightarrow{a} s_1'; s_2}$

Seq-2 $\quad \dfrac{\surd(s_1) \text{ and } s_2 \xrightarrow{a} s_2'}{s_1; s_2 \xrightarrow{a} s_2'}$

Ref-1 $\quad \dfrac{s \xrightarrow{b} s'}{s[a \rightsquigarrow Q] \xrightarrow{b} s'[a \rightsquigarrow Q]} \quad a \neq b$

Ref-2 $\quad \dfrac{s \xrightarrow{a} s' \text{ and } Q \xrightarrow{a'} s_1}{s[a \rightsquigarrow Q] \xrightarrow{a'} (*s_1); s'[a \rightsquigarrow Q]}$

Rec $\quad \dfrac{P[rec\, x.P/x] \xrightarrow{a} s}{rec\, x.P \xrightarrow{a} s}$

Star $\quad \dfrac{s \xrightarrow{a} s'}{*s \xrightarrow{a} *s'}$

A-Syn $\quad \dfrac{s_1 \xrightarrow{a} s_1'}{s_1 \parallel_A s_2 \xrightarrow{a} s_1' \parallel_A s_2 \text{ and } s_2 \parallel_A s_1 \xrightarrow{a} s_2 \parallel_A s_1'} \quad a \notin A \wedge \mathbf{ab}(s_2)$

Syn $\quad \dfrac{s_1 \xrightarrow{a} s_1' \text{ and } s_2 \xrightarrow{a} s_2'}{s_1 \parallel_A s_2 \xrightarrow{a} s_1' \parallel_A s_2' \text{ and } s_2 \parallel_A s_1 \xrightarrow{a} s_2' \parallel_A s_1'} \quad a \in A \wedge \mathbf{ab}(s_1) \wedge \mathbf{ab}(s_2)$

We'd like to comment on some special rules as follows: The rule Nd says that only two processes in $\mathcal{P}$ can be performed nondeterministically, the other cases are impossible by the definition of $\mathcal{P}^*$. The rule Ref-2 states that the residual s_1 of Q is non-interruptible. The rule Star says that $*s$ behaves like s, but the reached state is still concrete (if not properly terminated). The rule A-Syn gives priority to the concrete component. At any time, if a concrete process is in parallel with an abstract process, the latter has to remain idle till the former finishes the executing. Observe that there is no way to reach a state where both components are concrete, starting from an initial abstract state (in fact, such a state would not be well-formed). Moreover, if both components are abstract, the rule allows any of them to proceed first. The rule Syn states that

only two abstract processes can communicate each other. The communication between a concrete process and another process may destroy the atomicity of the refinement. In fact, it is impossible to reach a state where a concrete process synchronizes with another process from an initial abstract state. The other rules can be conceived as usual. The above rules guarantee that the execution of the refinement Q is not only to be non-interruptible, but also to be either executed completely, or not at all.

In the following, we investigate the notion of strong bisimulation on $\mathcal{P}^*$.

Definition 3. – *A binary symmetric relation R over the closed terms of $\mathcal{P}^*$ is a strong bisimulation if for any $(s_1, s_2) \in R$*
- *$\surd(s_1)$ iff $\surd(s_2)$; and*
- *for any $a \in Act$, $s_1 \stackrel{a}{\rightarrow} s_1'$, there exists s_2' s.t. $s_2 \stackrel{a}{\rightarrow} s_2'$ and $(s_1', s_2') \in R$.*

– *s_1 and s_2 are strong bisimilar, denoted by $s_1 \sim s_2$, if and only if there exists a strong bisimulation R such that $(s_1, s_2) \in R$.*

– *Let $E, F \in \mathcal{P}^*$ and $fn(E) \cup fn(F) \subseteq \{x_1, \ldots, x_n\}$. Then $E \sim F$ iff for any closed terms $s_1, \ldots, s_n$, $E\{s_1/x_1, \ldots, s_n/x_n\} \sim F\{s_1/x_1, \ldots, s_n/x_n\}$.*

According to the above semantics, it is easy to show that

Lemma 1. *For any closed term $s \in \mathcal{P}^*$, $s \sim *s$.*

Because a concrete process has a priority in parallel with an abstract process, $\sim$ is not preserved by $\|_A$. For example, $a_1; a_2 \sim a[a \rightsquigarrow a_1; a_2]$, but $(a_1; a_2) \|_{\{\}} b \not\sim a[a \rightsquigarrow a_1; a_2] \|_{\{\}} b$. However, once we strengthen Definition 3 by adding the following condition:

- $\mathbf{ab}(s_1)$ iff $\mathbf{ab}(s_2)$,

then the resulting largest bisimulation, denoted by $\sim_{\mathbf{ab}}$, is a congruence relation over $\mathcal{P}^*$. Besides, obviously, $\sim_{\mathbf{ab}}$ is a proper subset of $\sim$. That is,

Theorem 1. *$\sim_{\mathbf{ab}}$ is a congruence over $\mathcal{P}^*$ and $\sim_{\mathbf{ab}} \subset \sim$.*

Convention: From now on, we use $P, Q, \ldots$ to stand for processes in $\mathcal{P}^*$.

3 Fixpoint Logic with Chop (FLC)

FLC is an extension of the modal μ-calculus by introducing the chop operator ";", which can express non-regular properties [16]. It is therefore strictly more powerful than the μ-calculus, since [7,9] proved that only regular properties can be defined in the μ-calculus. For our purpose, we modify FLC [16] slightly.

Let $X, Y, Z, \ldots$ range over an infinite set *Var* of *variables*, *true* and $false$ be two propositional constants as usual, and $\surd$ be another one that is used to indicate if a process is terminated.

The formulae of FLC are generated according to the following grammar:

$$\phi ::= true \mid false \mid \surd \mid \tau \mid X \mid [a] \mid \langle a \rangle \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1; \phi_2 \mid \mu X.\phi \mid \nu X.\phi$$

where $X \in Var$ and $a \in Act$[1].

[1] In [16], τ is called *term*.

In the sequel, we use $\boxed{\text{a}}$ to stand for $\langle a\rangle$ or $[a]$, p for $true, false$ or $\surd$, and σ for ν or μ, $Act(\phi)$ for all actions that occur in ϕ.

As in the modal μ-calculus, the two *fixpoint operators* μX and νX bind the respective variable X and we will apply the usual terminology of *free* and *bound occurrences* of a variable in a formula, *closed* and *open* formulae etc. $fn(\phi)$ denotes the variables that have some free occurrence in ϕ and $bf(\phi)$ stands for the variables that have some bound occurrence in ϕ. X is said *guarded* in ϕ if each occurrence of X is in a sub-formula ϕ preceded with $\boxed{\text{a}}$ or p. If all variables in ϕ are guarded, then ϕ is called *guarded*.

FLC is interpreted over a given labelled transition system $T = (\mathcal{S}, A, \rightarrow)$, where $\mathcal{S} \subseteq \mathcal{P}^*$, $A \subseteq Act$, and $\rightarrow \subseteq \mathcal{S} \times A \times \mathcal{S}$. A formula is interpreted as a *monotonic predicate transformer*, which is simply a mapping $f : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$ that is monotonic w.r.t. the inclusion ordering on $2^{\mathcal{S}}$. We use $\mathrm{MPT_T}$ to represent all these monotonic predicate transformers over $\mathcal{S}$. $\mathrm{MPT_T}$, together with the inclusion ordering defined by

$$f \subseteq f' \text{ iff } f(\mathcal{A}) \subseteq f'(\mathcal{A}) \text{ for all } \mathcal{A} \subseteq \mathcal{S},$$

forms a complete lattice. We denote the join and meet operators by $\sqcup$ and $\sqcap$. By Tarski-Knaster Theorem, the least and greatest fixed points of monotonic functions: $(2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}) \rightarrow (2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}})$ exist. They are used to interpret fixed point formulae of FLC.

The meaning of $true$ and $false$ are interpreted in the standard way, i.e. by $\mathcal{S}$ and $\emptyset$ respectively. The meaning of $\surd$ is to map any subset of $\mathcal{S}$ to the subset of $\mathcal{S}$ which consists of all terminated processes in $\mathcal{S}$. Therefore, a process P meets $\surd$ iff $\surd(P)$. τ is interpreted as an identity. Because nil and δ have different behaviour in the presence of ;, they should be distinguished by FLC. To this end, $[a]$ is interpreted as a function that maps a set of processes $\mathcal{A}$ to the set in which each process is not terminated and any of the a-successors of the process must be in $\mathcal{A}$. This is different from its original interpretation in [16]. Therefore, according to our interpretation, $P \models [a]$ only if $\neg\surd(P)$. Whereas in [16], it is always valid that $P \models [a]$ for any $P \in \mathcal{P}^*$. So, it is easy to show that $nil \not\models \bigwedge_{a\in Act}[a]; false$, while $\bigwedge_{a\in Act}[a]; false$ is the characteristic formula of δ. The meaning of variables is given by an *environment* $\rho : var \rightarrow (2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}})$ that assigns variables to monotonic functions of sets to sets. $\rho[X \rightsquigarrow f]$ agrees with ρ except for associating f with X.

Definition 4. *Formally, given a labelled transition system $T = (\mathcal{S}, A, \rightarrow)$, the meaning of a formula ϕ, denoted by $\mathcal{C}^{\rho}_{T}(\phi)$, is inductively defined as follows:*

$$\begin{aligned}
\mathcal{C}^{\rho}_{T}(true)(\mathcal{A}) &= \mathcal{S}\\
\mathcal{C}^{\rho}_{T}(false)(\mathcal{A}) &= \emptyset\\
\mathcal{C}^{\rho}_{T}(\surd)(\mathcal{A}) &= \{P \mid P \in \mathcal{S} \wedge \surd(P)\}\\
\mathcal{C}^{\rho}_{T}(\tau)(\mathcal{A}) &= \mathcal{A}\\
\mathcal{C}^{\rho}_{T}(X) &= \rho(X)\\
\mathcal{C}^{\rho}_{T}([a])(\mathcal{A}) &= \{P \mid \neg\surd(P) \wedge \forall P' : P \xrightarrow{a} P' \Rightarrow P' \in \mathcal{A}\}
\end{aligned}$$

$$\begin{array}{rcl}
\mathcal{C}_T^\rho(\langle a\rangle)(\mathcal{A}) & = & \{P \mid \exists P' : P \xrightarrow{a} P' \wedge P' \in \mathcal{A}\} \\
\mathcal{C}_T^\rho(\phi_1 \wedge \phi_2)(\mathcal{A}) & = & \mathcal{C}_T^\rho(\phi_1)(\mathcal{A}) \cap \mathcal{C}_T^\rho(\phi_2)(\mathcal{A}) \\
\mathcal{C}_T^\rho(\phi_1 \vee \phi_2)(\mathcal{A}) & = & \mathcal{C}_T^\rho(\phi_1)(\mathcal{A}) \cup \mathcal{C}_T^\rho(\phi_2)(\mathcal{A}) \\
\mathcal{C}_T^\rho(\phi_1;\phi_2) & = & \mathcal{C}_T^\rho(\phi_1) \cdot \mathcal{C}_T^\rho(\phi_2) \\
\mathcal{C}_T^\rho(\mu X.\phi) & = & \sqcap\{f \in \mathrm{MPT_T} \mid \mathcal{C}_T^{\rho[X\rightsquigarrow f]}(\phi) \subseteq f\} \\
\mathcal{C}_T^\rho(\nu X.\phi) & = & \sqcup\{f \in \mathrm{MPT_T} \mid \mathcal{C}_T^{\rho[X\rightsquigarrow f]}(\phi) \supseteq f\}
\end{array}$$

where $\mathcal{A} \subseteq \mathcal{S}$, and $\cdot$ stands for the composition operator over functions.

A process P is said to satisfy ϕ iff $P \in \mathcal{C}_T^\rho(\phi)(\mathcal{S})$ for some environment ρ, denoted by $P \models \phi$. $\phi \Rightarrow \psi$ means that $\mathcal{C}_T^\rho(\phi)(\mathcal{A}) \subseteq \mathcal{C}_T^\rho(\psi)(\mathcal{A})$ for any T and $\mathcal{A} \subset \mathcal{S}$ and any ρ. $\phi \Leftrightarrow \psi$ means $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$. The other notations can be defined in a standard way.

Convention: In the sequel, we assume the binding precedence among the operators of the logic as follows: "$;$" > "$\vee$" = "$\wedge$" > "$\nu X.$" = "$\mu X.$" > "$\Rightarrow$" = "$\Leftrightarrow$".

Many properties of FLC have been shown in [16], e.g., FLC is strictly more expressive than the μ-calculus since context-free processes can be characterized by it; FLC is decidable for finite-state processes, undecidable for context-free processes; the satisfiability and validity of FLC are undecidable; FLC does not enjoy the finite-model property and so on.

[11] proved that FLC has the tree model property[2], i.e.,

Theorem 2. *Given $P, Q \in \mathcal{P}^*$, and $P \sim Q$, then for any closed ϕ, $P \models \phi$ iff $Q \models \phi$.*

Given a formula ϕ, we define its beginning atomic sub-formulae, denoted by $\mathrm{FSub}(\phi)$, as:

$$\mathrm{FSub}(\phi) \mathrel{\widehat{=}} \begin{cases}
\{\phi\} & \text{if } \phi = p, X, \boxed{\mathrm{a}} \text{ or } \tau \\
\mathrm{FSub}(\phi_1) \cup \mathrm{FSub}(\phi_2) & \text{if } \phi = \phi_1 \wedge \phi_2 \text{ or } \phi = \phi_1 \vee \phi_2 \\
\mathrm{FSub}(\phi_1) & \text{if } \phi = \phi_1;\phi_2 \text{ and } \phi_1 \not\Leftrightarrow \tau \\
\mathrm{FSub}(\phi_2) & \text{if } \phi = \phi_1;\phi_2 \text{ and } \phi_1 \Leftrightarrow \tau \\
\mathrm{FSub}(\phi_1) & \text{if } \phi = \sigma X.\phi_1
\end{cases}$$

Symmetrically, we define its ending atomic sub-formulae, denoted by $\mathrm{ESub}(\phi)$, as:

$$\mathrm{ESub}(\phi) \mathrel{\widehat{=}} \begin{cases}
\{\phi\} & \text{if } \phi = p, X, \boxed{\mathrm{a}} \text{ or } \tau \\
\mathrm{ESub}(\phi_1) \cup \mathrm{ESub}(\phi_2) & \text{if } \phi = \phi_1 \wedge \phi_2 \text{ or } \phi = \phi_1 \vee \phi_2 \\
\mathrm{ESub}(\phi_1) & \text{if } \phi = \phi_1;\phi_2 \text{ and } \phi_2 \Leftrightarrow \tau \\
\mathrm{ESub}(\phi_2) & \text{if } \phi = \phi_1;\phi_2 \text{ and } \phi_2 \not\Leftrightarrow \tau \\
\mathrm{ESub}(\phi_1) & \text{if } \phi = \sigma X.\phi_1
\end{cases}$$

Example 1. $\mathrm{FSub}(\langle a\rangle;\langle b\rangle \wedge [c];\langle e\rangle;[f]) = \{\langle a\rangle, [c]\}$, whereas $\mathrm{ESub}(\langle a\rangle;\langle b\rangle \wedge [c];\langle e\rangle;[f]) = \{\langle b\rangle, [f]\}$. ⊣

[2] The proof for the tree model property of FLC in [11] still works in our case.

When we say that $\surd$ only occurs at the end of ϕ it means that $\surd$ can only be in $\mathrm{ESub}(\phi)$ as a sub-formula of ϕ and can not appear elsewhere in the formula.

Definition 5. *A formula ϕ is called* existential formula *if for any $a \in Act$, $[a] \notin \mathrm{FSub}(\phi)$. We use $\mathcal{EF}$ to stand for the set of existential formulae. Dually, a formula ϕ is called* universal formula *if for any $a \in Act$, $\langle a\rangle \notin \mathrm{FSub}(\phi)$. We use $\mathcal{UF}$ to stand for the set of universal formulae. For technical reasons, we stipulate that $\tau \notin \mathcal{UF}$. A formula ϕ is called a property formula if $\phi \Leftrightarrow \phi_1 \wedge \phi_2$, where $\phi_1 \in \mathcal{EF}$ and $\phi_2 \in \mathcal{UF}$. The set of property formulae is denoted by $\mathcal{PF}$.*

For $\mathcal{EF}$ and $\mathcal{UF}$, we have

Theorem 3. *$\mathcal{EF}$ and $\mathcal{UF}$ are closed under all operators of the logic. I.e., $\phi\, op\, \varphi \in \mathcal{EF}(\mathcal{UF})$ and $\sigma X.\phi \in \mathcal{EF}(\mathcal{UF})$ if $\phi, \varphi \in \mathcal{EF}(\mathcal{UF})$, for any ϕ and φ, where $op \in \{\vee, \wedge, ;\}$.*

4 Towards Hierarchical Specification

As the complexity of reactive system designs becomes overwhelming very quickly, methods which allow to develop designs in a hierarchical fashion must be supported by the design formalisms employed. Such methods allow to develop a design at different levels of abstraction thereby making the development procedure more transparent and thus tractable: Most likely, a developer first divides the intended (complex) design into various "sub-designs" to capture the abstract overall structure of the complete design. Subsequently, the sub-designs will be developed by enriching them step by step with details. This is the design technique usually encountered in practice, see e.g. in [18]. In process algebraic settings, action refinement as introduced in Section 2 supports the hierarchical design.

In [15], we investigated the issue how to provide such a technique in a logical framework. To this end, a refinement mapping is defined by substituting the property of the refinement of an abstract action a for the modalities $\langle a\rangle$ and $[a]$ in a high-level specification and producing a lower-level specification. However, in [15], we only consider the case when all specifications are represented by some formulae in the subset $\mathcal{NF}$ of FLC called normal form formulae, which essentially correspond to the μ-calculus with τ, and the properties of refinements by some formulae without fixpoint operator in the subset. This is because in [15] we concentrated on the simple case to refine an abstract action by a finite process. Here, we consider the issue in general, i.e., refining an abstract action by any process. To this end, we adopt FLC itself as specification language, instead of $\mathcal{NF}$. This is because after refining a formula in $\mathcal{NF}$ with a property for a recursive process, the resulting formula may not be in $\mathcal{NF}$ any more. For example, suppose that $\langle a\rangle\phi \in \mathcal{NF}$ and a is refined by a process with the property $\nu X.\langle a'\rangle X \wedge \langle c'\rangle$. By our definition, the refined specification is $(\nu X.\langle a'\rangle X \wedge \langle c'\rangle)\phi$. It is easy to prove that there exists no $\varphi \in \mathcal{NF}$ such that φ is equivalent to the specification.

In a logical framework, actions are addressed as modalities and descriptions of systems are represented by formulae. In most of modal logics, there are two kinds of modalities, i.e. $\langle a\rangle$ and $[a]$ which are used to express existential and universal properties respectively. As discussed in [15], a refinement mapping should be property-preserving, i.e. an existential property should be refined to an existential property and similarly for the other properties. Otherwise, the mapping is meaningless since it's impossible to establish a correspondence between action refinement for models and action refinement for specifications. For example, $P \widehat{=} a; b + a; c \models \langle a\rangle; \langle b\rangle$, $a_1; a_2 \models [a_1]; \langle a_2\rangle$, but $P[a \rightsquigarrow a_1; a_2] \not\models ([a_1]; \langle a_2\rangle); \langle b\rangle$, since in the high-level specification, $\langle a\rangle; \langle b\rangle$ is an existential property, however its refinement becomes a universal property.

To ensure that the mapping is property-preserving, we partition the property ψ of the refinement of a into two parts: an existential property ψ_1 and an universal property ψ_2 i.e. $\psi \in \mathcal{PF}$. $[a]$ will be replaced by ψ_2, and $\langle a\rangle$ will be replaced by ψ_1. This is justified by the result shown in [4] that any property can be presented as the intersection of a liveness property and a safety property in branching temporal logics. So, $\mathcal{PF}$ is powerful enough to define the properties of reactive systems.

Therefore, we define the refinement mapping as follows:

Definition 6. *Suppose ϕ is a high-level specification, a is an abstract action to be refined, and $\psi \widehat{=} \psi_1 \wedge \psi_2 \in \mathcal{PF}$ is the description of the refinement of a, where $\psi_1 \in \mathcal{EF}$ and $\psi_2 \in \mathcal{UF}$. We define the refinement mapping, denoted by $\Omega(\phi, \psi, a)$, as follows:*

$$\Omega(\phi, \psi, a) \mathrel{\widehat{=}} \phi\{\psi_1\{\tau/\surd\}/\langle a\rangle, \psi_2\{\tau/\surd\}/[a]\},$$

where $\phi\{\psi/\chi\}$ means to substitute ψ for each occurrence of χ in ϕ, with $\chi \in \{X, \surd, \langle a\rangle, [a]\}$.

According to the above definition, it is easy to get the following results.

Lemma 2. *Suppose X does not occur in ψ. Then*

$$\Omega(\phi_1\{\phi_2/X\}, \psi, a) \Leftrightarrow \Omega(\phi_1, \psi, a)\{\Omega(\phi_2, \psi, a)/X\}.$$

Lemma 3. **(1)** *If $\phi \Leftrightarrow \phi'$ then $\Omega(\phi, \psi, a) \Leftrightarrow \Omega(\phi', \psi, a)$;*
(2) *If $\psi \Leftrightarrow \psi'$ and $\surd$ only occurs at the ends of ψ and ψ', then $\Omega(\phi, \psi, a) \Leftrightarrow \Omega(\phi, \psi', a)$.*

Theorem 4 (Applicability). *If $\phi \in FLC$ and $\psi \in \mathcal{PF}$, then $\Omega(\phi, \psi, a) \in FLC$; If $\phi, \psi \in \mathcal{PF}$, then $\Omega(\phi, \psi, a) \in \mathcal{PF}$.*

Here, we further study the example of a salesman that is firstly presented in [6] and has been investigated in [15] to demonstrate how to employ our approach to hierarchically specify a complex systems.

Example 2. Suppose that a salesman has to go by car from his office in Paris to another office in London and work there for some time, and then has to go back to Paris repeatedly. He takes a hovercraft to cross the Channel.

So, the top-most specification of the system can be represented as follows:

$$\phi \mathrel{\widehat{=}} \nu X. \left(\begin{array}{c} \langle \text{leave\_Paris} \rangle; [\text{fr\_thr\_Channel}]; \langle \text{arrive\_in\_London} \rangle; \langle \text{work} \rangle; \\ \langle \text{leave\_London} \rangle; [\text{gb\_thr\_Channel}]; \langle \text{arrive\_in\_Paris} \rangle; X \end{array} \right),$$

where the actions "work" and "x_thr_Channel" will be refined subsequently.

The job of the salesman in London is to contact repeatedly some of his customers by phone, or to meet some of them in his office to discuss something. Therefore, we can refine the action "work" by a process that meets the following property:

$$\psi_1 \mathrel{\widehat{=}} \nu X.(\langle \text{contact\_Customers} \rangle \vee \langle \text{meet\_Customers} \rangle); X \wedge \langle \text{finish\_Work} \rangle.$$

Meanwhile, we can describe "x_thr_Channel" in more detail. There are two platforms lying on the two sides of the Channel respectively that take charge of the hovercraft. At the beginning, one of them loads the salesman's car, then arranges the hovercraft to depart. Then the hovercraft crosses through the Channel. After the hovercraft arrives at the opposite side, the other platform unloads the car. Hence, "x_thr_Channel" can be enriched as follows:

$$\psi_x \widehat{=} [\text{x\_load}]; [\text{x\_departure}]; \langle \text{cross\_Channel} \rangle; \langle \overline{\text{x}}\text{\_arrival} \rangle; \langle \overline{\text{x}}\text{\_unload} \rangle \wedge \mathit{true}.$$

Furthermore, we can refine "x_departure" by a process with the property

$$\psi_2 \widehat{=} [\text{finish\_loading}]; \langle \text{engine\_on} \rangle; \langle \text{bye\_bye} \rangle \wedge \text{true},$$

where finish_loading signals the end of loading, and cross_Channel by a process with the property

$$\begin{array}{l} \psi_3 \mathrel{\widehat{=}} \mathit{true} \wedge \langle \text{sit\_down} \rangle; \\ \qquad (\nu \text{X}.(\langle \text{newspaper} \rangle \vee \langle \text{tea} \rangle \vee \langle \text{coffee} \rangle); \text{X} \wedge \langle \text{keep\_idle} \rangle); \langle \text{stand\_up} \rangle. \end{array}$$

So, the specification for the final system can be represented by

$$\Omega(\Omega(\phi, \Omega(\Omega(\psi_x, \psi_2, \text{x\_departure}), \psi_3, \text{cross\_Channel}), \text{x\_thr\_Channel}), \psi_1, \text{work}),$$

where $x \in \{fr, gb\}$, and if $x = fr$ then $\overline{x} = gb$ else $\overline{x} = fr$.

It is obvious that we can not refine "work" and "cross_Channel" by some processes that satisfy ψ_1 and ψ_3 respectively in [15] because on the one hand, the resulting specification is no longer in $\mathcal{NF}$; on the other hand, "work" and "cross_Channel" both are needed to be refined by some processes with possibly infinite behaviour. ⊣

5 Relating Hierarchical Specification to Hierarchical Implementation of a Large System

In this section, we establish a correspondence presented by the Refinement Theorem below between hierarchical specification and hierarchical implementation

of a complex system. It states that if $Q \models \psi;\surd$, $P \models \phi$ and some syntactical conditions hold, then $P[a \rightsquigarrow Q] \models \Omega(\phi, \psi, a)$. This result supports 'a priori' verification. In the development process we start with $P \models \phi$ and either refine P and obtain automatically a (relevant) formula that is satisfied by $P[a \rightsquigarrow Q]$. Or, we refine ϕ using $\Omega(\phi, \psi, a)$ and obtain automatically a refined process $P[a \rightsquigarrow Q]$ that satisfies the refined specification. Of course such refinement steps may be iterated.

In order to ensure the Refinement Theorem is valid, the following syntactical conditions are necessary:

Above of all, it is required that $(Act(P) \cup Act(\phi)) \cap (Act(Q) \cup Act(\psi)) = \emptyset$, because of the following considerations:

- **(i)** As far as action refinement for models is concerned, no deadlock will be introduced or destroyed;
- **(ii)** no unsatisfaction between $P[a \rightsquigarrow Q]$ and $\Omega(\phi, \psi, a)$ will be caused because ϕ involves Q. For instance, let $P \widehat{=} a; b$, $\phi \widehat{=} [a]; \langle b\rangle \wedge [c]; \langle d\rangle$, $Q \widehat{=} c; e$ and $\psi \widehat{=} [c]; \langle e\rangle$. It is obvious that $P \models \phi$ and $Q \models \psi;\surd$, but $P[a \rightsquigarrow Q] \not\models \Omega(\phi, \psi, a)$;
- **(iii)** Symmetrically, no unsatisfaction between $P[a \rightsquigarrow Q]$ and $\Omega(\phi, \psi, a)$ will be caused because ψ involves P. For example, let $P \widehat{=} a; b + b; a$, $\phi \widehat{=} [a]; \langle b\rangle$, $Q \widehat{=} c; e$ and $\psi \widehat{=} [c]; \langle e\rangle \wedge [b]; \langle d\rangle$. It is obvious that $P \models \phi$ and $Q \models \psi;\surd$, but $P[a \rightsquigarrow Q] \not\models \Omega(\phi, \psi, a)$.

It is clear that this condition can guarantee the above three requirements.

Besides, it's possible that ψ only describes partial executions of Q, so the refined specification may not be satisfied by the refined system. For example, it's obvious that $a; b + a; c \models \langle a\rangle; \langle b\rangle$ and $a_1; a_2 \models \langle a_1\rangle$, but $(a; b + a; c)[a \rightsquigarrow a_1; a_2] \not\models \langle a_1\rangle; \langle b\rangle$. In order to solve such a problem, we require that ψ describes full executions of Q, i.e., $Q \models \psi;\surd$. Normally, we only consider to refine an abstract action a by a normed process Q, i.e., for any derivative Q' of Q, Q' may terminate in finite steps. If so, for any given Q and $\psi \in \mathcal{PF}$ with $Q \models \psi$, the above requirement can be satisfied by constructing φ as $\psi; (\mu X.(\bigvee_{a \in Act} \langle a\rangle); X \vee \tau)$ instead of ψ. It is clear that $\varphi \in \mathcal{PF}$, $P \models \psi$ iff $P \models \varphi$ for each $P \in \mathcal{P}^*$, and $Q \models \varphi;\surd$. Therefore, in most cases, the above constraint does not give rise to any restriction to the applications of the theorem.

Finally, it is possible that $\surd$ as a sub-formula of ψ makes the sub-formulae following it with ; no sense during calculating the meaning of ψ, but the sub-formulae play a nontrivial role during interpreting $\Omega(\phi, \psi, a)$. E.g. $a'; nil \models \langle a'\rangle;\surd; [a']; \langle b'\rangle$ and $a; a; c \models \langle a\rangle; \langle a\rangle; \langle c\rangle$, but

$$(a; a; c)[a \rightsquigarrow a'; nil] \not\models (\langle a'\rangle; \tau; [a']; \langle b'\rangle); (\langle a'\rangle; \tau; [a']; \langle b'\rangle); \langle c\rangle.$$

So, we require that $\surd$ only can appear at the end of ψ as a sub-formula. In fact, such a requirement is reasonable because all formulae can be transformed to such kind of forms equivalently because $p; \phi \Leftrightarrow p$.

Now, we can represent our Refinement Theorem as follows:

Theorem 5 (Refinement Theorem).
If $(Act(P) \cup Act(\phi)) \cap (Act(\psi) \cup Act(Q)) = \emptyset$, $Q \models \psi; \surd$ *and* $P \models \phi$, *then* $P[a \rightsquigarrow Q] \models \Omega(\phi, \psi, a)$, *where* $\psi \in \mathcal{PF}$ *and* $\surd$ *only occurs at the end of* ψ.

In order to demonstrate how to apply the Refinement Theorem to verify a complex system hierarchically, we continue Example 2.

Example 3. As explained in Example 2, at the top level, we can implement the system as:

$$\text{Sys} \mathrel{\widehat{=}} \text{fr\_Channel} \parallel_{\{\text{fr\_thr\_Channel}\}} \text{Salesman} \parallel_{\{\text{gb\_thr\_Channel}\}} \text{gb\_Channel}.$$

Where $\text{x\_Channel} \mathrel{\widehat{=}} rec\, y.\text{x\_thr\_Channel}; y$, and

$$\begin{aligned}\text{Salesman} \mathrel{\widehat{=}} rec\, x.&\text{leave\_Paris}; \text{fr\_thr\_Channel}; \text{arrive\_in\_London};\\ &\text{work}; \text{leave\_London}; \text{gb\_thr\_Channel}; \text{arrive\_in\_Paris}; x.\end{aligned}$$

It's obvious that $\text{Sys} \models \phi$.
Then, we can refine "work" by Subsys_1 which is defined by

$$\text{Subsys}_1 \mathrel{\widehat{=}} rec\, x.((\text{contact\_Customers} + \text{meet\_Customers}); x + \text{finish\_Work}).$$

It's obvious that $\text{Subsys}_1 \models \psi_1; \surd$.
Then, "x_thr_Channel" can be implemented by
$\text{Subsys}_\text{x} \mathrel{\widehat{=}} \text{x\_load} || _{\{\text{x\_load}\}} \text{Channel}$,

$$\begin{aligned}\text{where Channel} \mathrel{\widehat{=}} \text{fr\_Platform} &\parallel_{\{\text{fr\_arrival,fr\_departure}\}} \text{Hovercraft}\\ &\parallel_{\{\text{gb\_arrival,gb\_departure}\}} \text{gb\_Platform},\end{aligned}$$

$$\begin{aligned}\text{where Hovercraft} &\mathrel{\widehat{=}} \text{fr\_departure;cross\_Channel; gb\_arrival} +\\ &\quad\text{gb\_departure;cross\_Channel; fr\_arrival},\\ \text{x\_Platform} &\mathrel{\widehat{=}} \text{x\_load;x\_departure} + \text{x\_arrival;x\_unload}.\end{aligned}$$

It's easy to show that $\text{Subsys}_\text{x} \models \psi_x; \surd$.
Furthermore, we can refine "x_departure" by Subsys_2 and "cross_Channel" by Subsys_3, where,

$$\begin{aligned}\text{Subsys}_2 &\mathrel{\widehat{=}} \text{finish\_loading; engine\_on; bye\_bye},\\ \text{Subsys}_3 &\mathrel{\widehat{=}} \text{sit\_down}; rec\, x.([(\text{coffee} + \text{tea})||_{\{\}}\text{newspaper}]; x + \text{keep\_idle}); \text{stand\_up}.\end{aligned}$$

Certainly, $\text{Subsys}_2 \models \psi_2; \surd$ and $\text{Subsys}_3 \models \psi_3; \surd$.
The final system is obtained as:

$$\begin{aligned}&\text{Sys}\,[\text{work} \rightsquigarrow \text{Subsys}_1,\\ &\quad \text{x\_thr\_Channel} \rightsquigarrow \text{Subsys}_\text{x}[\begin{matrix}\text{x\_departure} \rightsquigarrow \text{Subsys}_2,\\ \text{cross\_Channel} \rightsquigarrow \text{Subsys}_3\end{matrix}]],\end{aligned}$$

where $\text{x} \in \{\text{fr}, \text{gb}\}$.
According to the Refinement Theorem, the final system satisfies the final specification. ⊣

6 Concluding Remarks

In this paper, we extend our previous work on combining hierarchical specification with hierarchical implementation of a complex system by allowing to refine an abstract action by an arbitrary process. Technically, we also greatly simplify our previous work such that our method can be more easily applied in practice. Furthermore, we also establish a correspondence between hierarchical specification and hierarchical implementation that supports 'a priori' verification in system design.

Similar results are shown in [10,14], but in their approaches, a refined specification is obtained from the original specification and the refinement Q, where Q is a finite process. Therefore, besides sharing the restriction of our previous work [15], certain interesting expected properties of the refined system cannot be derived using their approaches. What's more, we can show that their approaches can be seen as a special case of our method from a specification-constructing point of view. [2] discussed composing, refining specifications of reactive systems as some sound rules of a logic. [1] considered the problem given a low-level specification and a higher-level specification, how to construct a mapping from the former to the latter in order to guarantee the former implements the latter. Our refinement mapping Ω maps the abstract specification to the lower-level specification, i.e. we go the converse direction.

In our framework, composing specifications also can be dealt with, for example, supposing $P \models \phi;\surd$, and $\surd$ only occurs at the end of ϕ and $Q \models \psi$, we can get a composite specification like $\phi\{\tau/\surd\};\psi$ for the combined system $P;Q$.

In this paper, we use the standard interleaving setting, so we only consider the case of atomic action refinement for models because the standard bisimulation notion is not preserved by non-atomic action refinement in this setting. In fact, we believe our approach may be applied to the case of non-atomic action refinement, too, if an appropriate logic which is interpreted over some true concurrent settings such as event-structures is available. But it is still an open question how to establish such kind of logics.

Acknowledgements

The author wants to thank Prof. Mila Majster-Cederbaum and Harald Fecher, who joined the previous work for many fruitful discussions related to the topic. In particular, the author thanks Prof. Mila Majster-Cederbaum for going through the whole paper and for some critical comments on it which improve the presentation of this paper too much. The author also thanks Dr. Wu Jinzhao and some anonymous referees for their useful comments on this paper.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253-284, 1991.

2. M. Abadi and G. Plotkin. A logical view of composition and refinement. *Theoretical Computer Science*, 114:3-30, 1993.
3. L. Aceto and M. Hennessy. Termination, deadlock, and divergence. *Journal of ACM,* Vol. 39, No.1:147-187. January, 1992.
4. A. Bouajjani, J.C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. ICALP'91, LNCS 510, pp. 76-92.
5. G. Boudol. Atomic actions. *Bull. European Assoc. The. Comp. Sci.* 38:136-144.
6. P. Degano and R. Gorrieri, Atomic Refinement in Process Description Languages. TR 17-91 HP Pisa Center, 1991.
7. E.A. Emerson and C.S. Jutla. Tree automata, μ-calculus, and determinacy. In proc. of 33rd IEEE Symp. on Found. of Comp. Sci., pp.368-377, 1991.
8. R. Gorrieri and A. Rensink. Action refinement. *Handbook of Process Algebra*, Elsevier Science, 1047-1147. 2001.
9. D. Janin and I. Walukiewicz. On the expressive completeness of the propositional μ-calculus with respect to monadic second order logic. CONCUR'96, LNCS 1119, pp.263-277.
10. M. Huhn. Action refinement and properties inheritance in systems of sequential agents. CONCUR'96, LNCS 1119, pp. 639-654.
11. M. Lange and C. Stirling. Model checking fixed point logic with chop. FOSSACS'02, LNCS 2303, pp. 250-263.
12. M. Lange. Local model checking games for fixed point logic with chop. CONCUR'02, LNCS 2421, pp. 240-254.
13. M. Majster-Cederbaum and F. Salger. Correctness by construction: towards verification in hierarchical system development. SPIN'00, LNCS 1885, pp. 163-180.
14. M. Majster-Cederbaum and F. Salger. Towards the hierarchical verification of reactive systems. To appear in *Theoretical Computer Science*.
15. M. Majster-Cederbaum, N. Zhan and H. Fecher. Action refinement from a logical point view. VMCAI'03, LNCS 2575, pp.253-267.
16. M. Müller-Olm. A Modal Fixpoint Logic with Chop. STACS'99, LNCS 1563, pp. 510-520.
17. A. Rensink. *Models and Methods for Action Refinement.* PhD thesis, University of Twente, Enschede, Netherlands, Aug. 1993.
18. J. Sifakis. Research directions for concurrency. *ACM Computing Surveys,* 28(4):55. 1996.