A Design Methodology for Mobile Distributed Applications based on UNITY formalism and Communication-Closed Layering

Camelia Zlatea **Tzilla Elrad**

Computer Science Department,
Illinois Institute of Technology,
{ camelia, elrad }@charlie.iit.edu

ABSTRACT

Designing applications for today's mobile systems becomes an increasingly complex task. The execution of certain applications relies on location-dependent services, e.g. finding the shortest route or the nearest geographical configuration. However, for many other cases, location is transparent, e.g. querying the same database server. Mobile applications need to be adapted to disconnection or re-configuration of the underlying architecture or the environment. We are developing DUALITY a design framework for both types of mobile distributed applications. DUALITY enforces the separation of concerns of correctness - architecture independent specification and efficiency - mobility-aware mapping. The design model refines causality and concurrency properties. It coordinates transparency and adaptation for applications running on mobile distributed networks. Our approach relates mobility with existing concurrency design patterns and aims to reuse design knowledge of traditional distributed algorithms, therefore to reduce overall complexity.

KEYWORDS

Specification and Verification, Distributed Systems, Mobile Computing, Formal Methods.

1. INTRODUCTION

Mobile distributed systems rely on transient communication channels allowing short-lived interactions. Mobile computing characteristics include re-configurable hardware and software components, disconnected operations and location-aware services. One major issue when designing mobile distributed applications is the amount of details with respect to mobility concerns that should be modeled. The design approach should make a clear distinction between the problem, the algorithm, and the mobile distributed architecture, the implementation target. Balancing the issues of transparency and adaptation makes mobile application design an increasingly complex task. The execution of certain applications relies on location-dependent services, e.g. finding the shortest route or the nearest geographical configuration. However, for other cases, location is transparent, e.g. querying the same database server. Mobile applications need to be adapted to disconnection or re-configuration of the underlying architecture or the environment. The unavailability of the channel or the disconnection of the sender or the receiver, due to a lower battery power or a lower frequency bandwidth, should be no longer communication failures, but counted upon events. The tolerance to disconnection, the reaction to hardware/software resource availability, location or temporal awareness should be specified gradually from the earliest phases of the design.

Amadio [1] gives a design model for mobile agents and analyzes the effect of failures in the context of tolerance to disconnection and adaptation to resource availability. McCann and Roman [9] derive a UNITY-like methodology and address the problem of dynamically reconfiguring mobile distributed systems. Mobile components are programs with additional variables representing location. A form of dynamic composition, which extends UNITY's *union* and *superposition* with coordination constructs, specifies the interactions between mobile components.

In the following, we propose DUALITY, a design model for mobile distributed applications, which coordinates the issues of program correctness, implementation accuracy and efficiency. Our most important goal is to reduce the design complexity. We ensure the consistency between the specification and the architectural details by integrating formal methods and automated verification. In DUALITY, one performs the verification of the design correctness, hand in hand with the program transformations. We build the design methodology on UNITY's [3, 10, 11] notations and logic and on the principles and algebraic laws of communication closed layers (CCL) [4, 8, 13]. The choice is motivated by the availability of a programming notation and a temporal logic based proof systems, within the same framework. We claim that our design model can be also supported by partial-order temporal logic specifications like ISTL [12], or at some extent by CTL^* [2].

1.1 UNITY Overview. UNITY, the parallel programs design formalism, provides an implementation-independent specification notation and a proof system based on temporal logic. UNITY has the semantics of state-based transition systems, but uses a notation based on variables and assignments. A program is decoupled from its implementation and it derives much of its simplicity by abstracting away from the notion of control flow. Programs are mapped, then implemented on different types of architectures. A program has a section for variable declarations, a section specifying the initial values for the variables (the *INIT* predicate) and a finite set of assignments. A program is similar to a "bag" of assignments. Each assignment can be selected nondeterministically and eventually executed, and returned to the "bag". The execution model for UNITY programs is based on sequences of tuples (a tuple being a state). Each element of a sequence models an execution step of the program. A sequence denotes a possible run. A program can terminate if it reaches a *fixed point*, where no state transition is possible.
Synchronous and asynchronous actions are specified by assignments connected by the operator "||" and "[]"respectively. Assignments are selected for execution in a nondeterministic but fair manner. There is no notion of enabledness of a statement. This is equivalent to weak fairness [6]. In UNITY, the program is decoupled from its implementation and abstracts from the notion of control flow. A program can be mapped, then implemented on different types of architectures. The correctness of the UNITY program is independent of the target architecture and the manner in which the program is executed. However, efficiency must be evaluated with reference to the target architecture.

Notations:

s, t – UNITY assignments; *s.r* – the set of variables read by s; *s.w* - the set of variables written by s; *L, F*- UNITY programs; *L.a* - the set of all assignments, belonging to L; *L.r* - the set of all variables read by L *L.w* - the set of all variables written by L; *p* - a precondition that must be true before the execution of *s* and *q* – a postcondition, resulting after the execution of *s*. Proofs in UNITY are based on assertions of type {p} s {q}. A property P(p,q) applies globally to a program F, i.e. {p} F {q}, if and only if <∀s : s∈ F :: {p} s {q}>. For brevity, this paper focuses on three of the UNITY logic properties: *invariant* and *co*– safety properties and *leads-to* –progress property.

"*invariant p* ≡ (*INIT* ⇒ p) ∧ < {p} F {p} > . To illustrate, "*invariant* x>0" states that the variable x is positive in any state of the program execution.
"*p co q* ≡ < ∀s: s in F :: {p} s {q} >"; it signifies that if a state satisfying p occurs anywhere in the execution, the next state must satisfy q.
"p leads-to q" - signifies that once p becomes true q will be true eventually. However, it cannot be asserted that p will remain true as long as q is not. To illustrate, "x=0 *leads-to* x>0", states that whenever x is equal to 0 during a program execution, then it will eventually become positive at some later state in the execution.

A *Fixed-Point* (*FP*) in a UNITY program, if one exists, is a program state such that the execution of any statement in that state leaves the state unchanged.

1.2 Communication Closed Layers. Communication closed layers principle [4] and algebraic laws [8] formalize the concept of weak sequential composition. The idea is that actions which are not dependent (unilateral or bilateral data dependent) do not have to wait for one another to proceed, even if they are composed sequentially. In the DUALITY framework, we relate UNITY model with the interleaving and the partial-order semantics. The interleaved execution of a UNITY program can be interpreted as a succession of a partial order executions. To define program structures, modules, we extend UNITY with programming abstractions that enable a direct representation of properties involving temporal and causal order. UNITY programs decomposition can be facilitated by clearly identifying the interface between program components, modules, layers, with the help of a set predicates. DUALITY does not consider that algebraic syntactic structure of the programming notation is too close to the target architecture. Much more, it is claimed that what leads to dependency, on one or other architecture, is due to certain language operations such as sequential operation. A model for concurrency is proposed. It allows distinction between the parallel composition [3] of processes F [] G, classical sequential composition F ; G and a weak sequential composition, called layer composition, F • G (inspired by Elrad and Francez [4] Communication-closed Layers (CCL) principle). For the *layer composition* F•G, any event *f* from F precedes any event *g* from G, with which is in conflict, in causal dependence. By comparison, *sequential composition*, denoted with "**;**", is a representation of the temporal dependence. Janssen, Poel and Zwiers in [8, 13] formulate the algebraic law of the CCL,

(F [] H) • (G [] K) = (F • G) [] (H • K)

providing that there is no conflict between the programs F and K and between H and G respectively.

This law is not valid if causal order, expressed by the layer composition, is replaced with temporal order, represented by the sequential composition. The communication closed layer principles enables a direct representation of properties involving temporal and causal order. Identifying causality of the events helps in bounding the nondeterminism. Serializability is a property that encompasses causal order and concurrency. It is very useful for program refinement, preserving mapping in the syntactic domain, for the purpose of efficient implementations on mobile distributed architectures.

The remainder of the paper is organized as follows. Section 2 presents the partial-order structures, as communication closed layers, defined in terms of UNITY semantics. DUALITY model is described in section 3. We explain how the principle of communication-closed layers correlates algorithmic computations and mobility concerns, hence coordination of *the specification* and *the mapping*. Section 4 illustrates a case of program transformation and design reuse based on a variant of All-Points Shortest Path Problem, our example to reason on coordinated design of mobile distributed systems. Conclusions and future work directions are summarized in the last section.

2. EXTENDING UNITY WITH PARTIAL ORDER BASED STRUCTURES

UNITY defines *union* as a nondeterministic parallel composition. The union of the programs S_1 and S_2 denoted S_1 [] S_2, has the following properties: $INIT(S_1 \; [] \; S_2) = INIT(S_1) \wedge INIT(S_2)$; $(S_1 \; [] \; S_2).a = S_1.a \cup S_2.a$. The main idea is that the safety of the composite program follows from the safety of its components. A shared-memory or a distributed program can be represented as a union: $F = S_1 \; [] \; S_2 \; [] \; .. \; [] \; S_n$. A distributed algorithm can still be described as a union composition. Program components communicate through channels (representing shared sequence variables) or perform independently. Nevertheless, union composition is static and limits an expressive mapping of the specification on mobile architectures. It does not reflect dynamic and adaptive relations between mobile components. The absence of control structures in UNITY has the drawback of making difficult to devise efficient implementations on mobile distributed architectures. This leads us to extend the UNITY notation for the mappings to provide certain forms of control structures, by isolating weak sequential phases and their interrelationship.

Property 1 Strong Enable s. If s labels an assignment, we define *s.event* a predicate that is true whenever s is selected and the guard is true (eventually this leads to a program state transition): *strong enable s* $\equiv$ *s.event*.

Property 2 Weak Enable s. If a selected statement s has false guard, *s.event* is false: *weak enable s* $\equiv \neg$*(s.event)*. (This is equivalent to the execution of a skip statement.).

Definition 1 Progress Condition. *L.event* denotes a predicate, which is true if exists an assignment $s \in L.a$, such that *s.event* is true (i.e., *strong enable s* holds). All the states where *L.event* holds make a consistent set of local states associated with the program L.

$L.event \equiv \langle \exists \; s : s \in L.a \wedge s.event \rangle$.

Definition 2 Enabling Condition. *L.init* denotes a predicate associated with the initial set of states, which precedes the selection and execution of any assignment belonging to L.a.

Definition 3 Fixed Point. If exists a *Fixed Point*, FP_L, for the program L this is a state predicate such that the execution of any statement $s \in L.a$ in that state leaves the state unchanged:

$FP_L \equiv \langle \forall s : s \in L.a \wedge p :: \{p\} \; s \; \{p\} \rangle$.

UNITY program structures are identified by the enabling conditions, progress conditions, the termination condition – the *Fixed-Point*. Now, we define the UNITY program structure in relation with all these conditions.

Definition 4 Group-of-Actions. A Group-of-Actions is a UNITY-like program structure L identified by the enabling condition, progress condition and respectively the fixed-point such that following properties hold:

1. *L.init* co (*L.event* $\vee$ *L.init*)
2. *L.event leads-to* FP_L

Property (1) states that the Group-of-Actions L can be activated initially only at its enabling condition. Property (2) states that the progress condition for the program L, causally precedes the eventual fixed point.

<u>Example:</u> A program F that uses Sieve algorithm to compute the first *count* = K prime numbers from a sequence of natural numbers *seq.i* ($1 \le i \le N$, $N > K$). F is a union composition $F = \langle [] \; j : 1 \le j \le N :: L_j \rangle$. The interfaces of the group of action L_j are specified by predicates: $L_j.init \equiv count = j - 1 \wedge j \le K \wedge seq.i = j < N$; $FP_{Lj} \equiv j < seq.i < N$; $L_j.event \equiv seq.i \ge j \wedge count \le K$;

Program L_j

initially count = j - 1 $\wedge$ j $\leq$ K $\wedge$ seq.i = j < N

assign

count, i := count +1, i+1 if seq.i = j

[] i := i +1 if (seq.i > j) $\wedge$ (seq.i mod j = 0)

[] seq.j+1, i := seq.i, i+1 if (seq.i > j) $\wedge$ (seq.i mod j $\neq$ 0)

end

We introduce new compositional abstractions based on Group-of-Actions that restrict nondeterminism and enable a direct representation of properties involving temporal and causal order. (Notations used Li and Lj groups of actions and s, t UNITY assignments.)

Definition 5 Mutual-Dependence.

a) "s mutual-dependent t" iff (s data-bilateral-dependent t) $\wedge$ (s temporal-bilateral-dependent t).

Example: s: x:=1 if y=0 [] t: y:=0 if x=1 where (s.w $\cap$ t.r $\neq \emptyset$) $\wedge$ (s.r $\cap$ t.w $\neq \emptyset$)) .

b) "Li mutual-dependent Lj" iff $\exists$ s$\in$ Li.a $\wedge$ $\exists$ t Lj.a, such that "s mutual-dependent t" holds.

Definition 6 Safe-Dependence

a) "s safe-dependent t" iff (s temporal-unilateral-dependent t) $\vee$ (s data-unilateral-dependent t).

Example: s: y:= x –1 [] t: x:=x+1 where s.r = { x } s.w = { y } t.r = t.w = { x } or always s.event precedes t.event.

b) "Li safe_dependent Lj" iff $\forall$ s $\in$ Li.a $\wedge$ $\forall$t $\in$ L_j.a, such that "s safe-dependent t" holds.

Definition 7 Communication-Closed Layer (CCL) Structure. A group of actions L_j (j = 1..k) is called a *CCL structure* in the context of a program F, if there exists a partitioning of the program F, F.a = $\cup$ L_j.a (j = 1..k), such that for any two group of actions L_j, L_{j+1} (j = 1..k-1), "L_j safe-dependence L_{j+1}".

Definition 8 Partial-Order Layer (POL) Structure. A *CCL structure* L_j (j = 1..k) is called a *POL structure* in the context of a program F, iff for any L_j (j = 1..k), the relation "mutual-dependence " is a transitive closure.

Lemma 1 CCL Progress/Safety Properties. If L_j (j = 1..k) is a *POL structure* in the context of a program F, there exists LI_j a predicate, such that:

a) "*invariant* LI_j" holds in all consistent local states (also called global snapshot [12]) determined by the executions of the program L_j.

b) "LI_j *leads-to* LI_{j+1}" and "$\neg$ (LI_{j+1} *co* LI_j)" holds for the program resulted by reunion of L_j and L_{j+1}.

Proof: From definition 7: a) for any POL structure L_j (j = 1..k), $\exists$ predicate LI_j : <$\forall$s, t $\in$ L_j.a , "s mutual dependent t"> $\equiv$ (L_j.*INIT* $\Rightarrow$ LI_j) $\wedge$ < { LI_j } F { LI_j } > $\equiv$ *invariant* LI_j ;

b) For any POL structures L_j and L_{j+1} (j = 1..k-1) $\exists$ predicate LI_j , "L_j safe-dependence L_{j+1}" $\Rightarrow$ "L_j temporal-unilateral dependent L_{j+1}" $\Rightarrow$ "LI_j *leads-to* LI_{j+1}". By contradiction, we demonstrate that "L_j safe-dependence L_{j+1}" $\Rightarrow$ "$\neg$ (LI_{j+1} *co* LI_j)"

Lemma 2 "CCL-POL Transformation".

a) Any *CCL structure* L can be represented as a safe dependent layer composition (denoted "•") of *POL structures*, L_i 1≤i<≤m : L $\equiv L_1 \bullet L_2 \bullet .. \bullet L_m$.

b) $\exists$ "*invariant LI*" the holds in all consistent local states determined by the executions of L.

Proof: a) Apply definition 7 and 8. b) For example: "*invariant LI*" = "*invariant* LI_1"$\vee$..$\vee$= "*invariant* LI_1

Lemma 3 "Union-CCL Transformation". Given a program F as a union composition: F = S_1 [] S_2 [] .. [] S_n. If there exists a partitioning of the program F, F.a = $\cup$ L_j.a (j = 1..k), such that any L_j (j = 1..k) is a *CCL structure*, then the union and the communication-closed layer compositions (denoted "•") are equivalent: $F = S_1 [] S_2 [] .. [] S_n \equiv L_1 \bullet L_2 \bullet .. \bullet L_k$

Discussion: Any program Lj represents an equivalence class on mutual-dependence relation. The union composition can be transformed into an equivalent communication-closed layer composition, providing that no mutual inter-layer, "conflict" communications occur during the execution of any program Lj (j=1,n). Therefore, the union composition F can be equivalently rewritten as a composition of communication-closed layers, based on the causal order of the events.

3 DUALITY DESIGN MODEL

Our strategy for designing mobile distributed programs consists of two cooperative phases.

i) The problem-oriented specification, "What?" is an architecture-independent representation of the computations. This is the premise to achieve transparency, for applications running in a mobile environment. Correctness properties apply globally to the transparent specification. We focus on two types of properties: Safety – something "bad" never happens, and Progress – something "good" eventually happens.

ii) The mapping, "How? When? Where?" an abstract description of how to execute a specification on a target mobile distributed architecture. Mapping represents architectural details, e.g. location awareness, adaptation to mobility constraints. Complexity and efficiency are major concerns of the mapping.

We introduce the *Meta-Program* as a consistent coordination of architecture-independent specifications and efficient implementations. A Meta-Program is a causal and concurrent, dual, composition of the adaptive mapping and the transparent specification. This notion extends Chandy and Misra's program-schema [3] for the area of mobile distributed application design.

Controlling the effect of nondeterminism is one problem that affects the design of mobile distributed applications. In a Meta-Program, there is a duality of causal and concurrent relations between specification and mapping events. The nondeterministic interleaving happens due to the asynchronous character of the mobile agents, static hosts and transient communication channels. Let consider two mobile agents, *a1* and *a2*, sending the messages b_{a1} and respectively b_{a2} to the same fixed host, *h*, in mailbox *B*. The representations of the two actions are labeled assignments: x: $B_h := b_{a1}$ and respectively y: $B_h: = b_{a2}$. The mapping expresses how the assignments are executed, by channel availability abstractions, e.g. m_x: *connected*(a_1,h) and respectively m_y: *connected(a2,h)*. We denote the nondeterministic interleaving of specification and mapping events with

Concurrent(m_x: *connected*(a_1,h), m_y: *connected(a2,h)* , x: $B_h := b_{a1}$, y: $B_h: := b_{a2}$) ; using a simplified notation: *Concurrent*(m_x, m_y, x, y) .

Under the interleaving semantics [12], a single execution of the Meta-Program is considered a totally ordered sequence of events. The semantics of the Meta-Program is given by its set of all possible total order executions: (x m_y), (m_y x), (x m_x), (m_x x), (y m_y), (m_y y), (y m_x), (m_x y).

The two total-orders of independent events, (x m_y) and (m_y x) are equivalent, and so are the total-orders (y m_y) and (m_y y). One calls the unordered sequence of the events x or m_y a partial-order. Under this semantics, a single execution is considered to be a partially-ordered sequence of events. The semantics of the Meta-Program, is given by the following six partial-order executions: (x m_y, m_y x), (x m_x), (m_x x), (y m_y), (m_y y), (y m_x, m_x y). Partial-order semantics makes a clear distinction between causality and concurrency, and allows for a more expressive specification. With partial-order relations, one can determine equivalent sequences of events. In this way, instead of analyzing all possible interleavings, we can only consider at least one representative from each equivalence class. This approach has two folds:

i) It allows for a simplified, less expensive and more efficient verification of the system's properties.

ii) It supports better decisions among the number of possible alternatives, consequently a reduced design complexity.

The semantics of *POL structures* and *CCL structures* allow the verification of the properties of the mobile concurrent systems, without exploring all possible interleavings of nondeterministic executions. One benefit is the capacity to manipulate the *layer composition* as a partial-order, mapping operator.

The Meta-Program is defined initially as: MP_S = *Concurrent*(m_x, m_y, x, y) ≡ m_x [] m_y[] x []y. For all executions of the Meta-Program, there is a causal order between the events, m_x and x and respectively m_y and y. Providing that there is no conflict between *x* and m_y and respectively *y* and m_x, the ordering between non-conflicting events can be ignored. Applying definitions 5, 6,7 and 8, we identify two *CCL structures*: the specification *S* = *Concurrent*(x, y) ≡ x [] y and the mapping M_S

$= Concurrent(m_x, m_y) \equiv m_x [] m_y$, two partial-orders in safe-dependent, causal relation: $MP_S = Causal(M_S, S)$.

The relation Concurrent(s,t) expresses the fact that the actions s and t can happened in any order, consequently mutual dependence is a concurrent relation: "s mutual-dependent t"≡ Concurrent(s,t). The relation Causal(s,t) expresses the data or temporal causality between actions s and t, therefore "s safe-dependent t"≡ Causal(s,t).

The Meta-Program can be expressed equivalently in terms of layer and union composition:

$$MP_S = Causal(M_S, S) = Causal(Concurrent(m_x, m_y), Concurrent(x, y)) \equiv M_S \bullet S_= (m_x [] m_y) \bullet (x [] y)$$

This design strategy leads us to the algebraic laws of communication closed layer, that allow for gradually adding details of temporal ordering to a Meta-Program that is already proven correct.

$$MP_S = Causal(Concurrent(m_x, m_y), Concurrent(x, y)) = Concurrent(Causal(m_x, x), Causal(m_y, y)) \equiv$$

$(m_x [] m_y) \bullet (x [] y) = (m_x \bullet x) [] (m_y \bullet y)$ this ca be re-formulated as,

$MP_S = MP_x [] MP_y = M_S \bullet S$, where MP_x and MP_y are Meta-Program components.

Next we give a generalization and formulate the DUALITY principle. With UNITY's *union* composition, "[]", we represent the nondeterminism, the concurrency among the components of specification or mapping: Let assume the an architecture independent, transparent specification:

$$S = Concurrent(S_1, S_2,\dots, S_n) \equiv S_1 [] S_2 [] \dots [] S_n$$

A possible mapping to a shared memory parallel architecture is a union of n communication transient channels, denoted

$$M_S = Concurrent(M_{S1}, M_{S2, \dots,} M_{Sn}) \equiv (M_{S1} [] M_{S2} [] .. [] M_{Sn}).$$

The *Meta-Program* expression is:

$$MP_S = Causal(M_S, S) \equiv M_S \bullet S$$

When designing a mobile system we should be able to represent in the mapping program the notions of transient communication channels and disconnected operations. An *adapter* is a new variable type added in UNITY to express mobility and location constraints. Mapping programs should have a self-adaptive behavior.

x = y *self-adapt* connected(location(x),location(y))
~ replicate(x) *self-adapt* disconnected(location(x),location(y))

Figure 3.1

The variables of type *adapter* express mobility constraints and location awareness. A reactive guard construction called *self-adapt* expresses the adaptation and the reaction of the mapping program to mobility constraints (Fig. 3.1). We reason about mobile distributed applications, based on a weak composition of communication closed layers, determined by the tolerance to disconnection. When components become disconnected, they are allowed to execute asynchronously. In a Meta-Program modeling a mobile distributed application, the mapping component is an adaptive layer that causally coordinates the events of the specification layer. The mapping program acts as a regulator that controls the specification. It can delay or inhibit basic interactions (assignments) or it can re-allow them, in accordance with location or mobility constraints. The *Meta-Program* is re-written:

$$MP_S = Causal(M_S, S) = Causal(Concurrent(M_{S1}, M_{S2,\dots,}M_{Sn}), Concurrent(S_1, S_2,\dots, S_n)).$$

We formulate in terms of UNITY the generalized communication closed layers algebraic laws [8,13] to allow transformations of the *Meta-Program* in forms that match shared memory or distributed architectures. Each transformation step is carried out in conjunction with a formal verification phase, to increase confidence in the design correctness and to allow earlier error detection. This method is highly applicable in the initial stages of the design. Assuming that L_{ij} and L_{kl} are *CCL structures* in the context of program F. If there is a causal dependence between Lij and Lkl then either i=k or j=l is satisfied, for $1 \le i, j, k, l \le n$. The generalized algebraic laws of communication closed layers are formulated as follows:

$$Concurrent(Causal(L_{11}, L_{12}, .., L_{1m}), Causal(L_{21}, L_{22}, .. , L_{2m}),..,Causal(L_{n1}, L_{n2}, .. , L_{nm})) =$$

$$Causal(Concurrent(L_{11}, L_{21}, .. , L_{n1}),Concurrent(L_{12}, L_{22}, .., L_{n2}), .. ,Concurrent(L_{1m}, L_{2m}, L_{nm}))$$

$$\equiv (L_{11} \bullet L_{12} \bullet .. \bullet L_{1m}) [] (L_{21} \bullet L_{22} \bullet .. \bullet L_{2m}) [] \ldots [] (L_{n1} \bullet L_{n2} \bullet .. \bullet L_{nm}) =$$

$$(L_{11} [] L_{21} [] .. [] L_{n1}) \bullet (L_{12} [] L_{22} [] .. [] L_{n2}) \bullet .. \bullet (L_{1m} [] L_{2m} [] .. [] L_{nm})$$

Applying the generalized laws of communication closed layers on the *Meta-Program* MP_S, we obtain the DUALITY principle in practice:

$$MP_S = Causal(M_S, S)$$

$$=Causal(Concurrent(M_{S1},M_{S2,\ldots,}M_{Sn}), Concurrent(S_1, S_2,\ldots, S_n))$$

$$= Concurrent(Causal(M_{S1}, S_1), Causal(M_{S2}, S_2), .., Causal(M_{Sn}, S_n))$$

$$= Concurrent(MP_{S1}, MP_{S2,\ldots,}MP_{Sn})$$

DUALITY principle formulated in terms of layer and union composition:

$$MP_S = M_S \bullet S = (M_{S1} [] M_{S2} [] .. [] M_{Sn}) \bullet (S_1 [] S_2 [] .. [] S_n) = (M_{S1} \bullet S1) [] (M_{S2} \bullet S2) [] .. [] (M_{Sn} \bullet Sn)$$

$$= MP_{S1} [] MP_{S2} [] .. [] MP_{Sn}$$

By transforming the initial *Meta-Program*, we coordinate mobility constraints with specific clusters of actions (assignments). Using the DUALITY model, one benefits from the design knowledge of a static algorithm (*S*) and applies it to the design of a correspondent mobile application (MP_S). *S* is a non-adaptive, transparent, architecture-independent specification represents a fragment from a program not aware of mobility. We add the mobility concerns, by synchronizing causally dependent components between specification and mapping. Specification of the initial, static computation can be reused. Mobility awareness is described by the mapping, the regulator layer M_S. Consistent coordination implies that if the architecture independent specification of a static application is S_i, the correspondent mobile application, MP_{Si}, can reuse S_i's design knowledge. DUALITY supports the derivation of efficient mobile distributed implementations from program specifications. Consequently, transparent and architecture-independent specifications can be transformed into forms that match particular mobile target architectures.

The correctness of the Meta-Program is deduced from the correctness of the architecture-independent specifications, preserved at each refinement step. More specifically, the correctness of the Meta-Program is inferred from the individually identified program modules (e.g., *CCL structures*).

Lemma 4 Safety of Duality. If *invariant* I holds in the union composition (MP_1 [] MP_2) and the same *invariant* I holds in the layers composition ($M_F \bullet F$), then the two forms are input/output equivalent: $MP_1 [] MP_2 \equiv M_F \bullet F$

Lemma 5 Progress-Safety of Duality. Given $MP_F = M_F \bullet F$, *invariant* I_{MF} holds in M_F, *invariant* I_F holds in F, then the formula (*invariant* I_{MF}) *leads-to* (*invariant* I_F) holds over the Meta-Program MP_F.

Discussion*:* I_{MF} is an assertion, on the global snapshot, associated with the layer M_F; I_F is also an assertion, on the global snapshot, associated with the layer F. No matter how the Meta-Program evolves, *if* I_{MF} is possible at some time, then I_F is also possible at a later time. This is a safety-progress property applied on the causally coordinated mapping and specification, the dual representation of causality and concurrency properties of the Meta-Program.

Related work has investigated manual proofs for communication closed layers [4, 15] and mechanized verification based on partial-order model checkers. Partial global snapshots in UNITY have been suggested in [10] as recording predicates that hold at some point in the computation.

4 EXAMPLE OF SPECIFICATION AND DESIGN REUSE

To illustrate our design model, let us take the case of a network of *n* mobile autonomous agents, sharing a distributed map of their current cell sites. (For instance, a number of intelligent Internet enabled wireless phones, equipped with processor and local memory.). If a channel is available between the location of t and the location of u, then the agents u and v can establish a two-way message communication. The mobile agents can update the distributed map and cooperate in finding the shortest distance between all pairs of agent locations. In order to design this mobile distributed application, we decide to reuse the design knowledge of the Floyd-Warshall algorithm. One starts with an implementation and architecture independent specification of the problem and addresses the issue of proving correctness for the problem solution. Next, one refines the initial solution for the case of a static distributed architecture, by preserving safety and liveness properties. This specification is then mapped on a target, distributed mobile architecture, typified by the reconfiguration of the moving elements. There are a number of details related to mobility that are stipulated in the mapping program:

1) Location dependent computations

2) Disconnected communication.

Finally, we reason on the correctness of the layered *meta-program* of specification and mapping, the solution to the mobile distributed application.

Notations:

Quantified union, asynchronous parallel composition - $<[]\ i : 1 \le i \le n : Si> = S1\ []\ ..\ []\ Si\ []..[]\ Sn$

Quantified synchronous parallel composition – $<\|\ i : 1 \le i \le n : Si> = S1\ \|\ ..\ \|\ Si\ \|..\|\ Sn$

Quantified layered composition – $<\bullet\ i : 1 \le i \le n : Li> = L1\ \bullet..\bullet\ Li\ \bullet..\bullet\ Ln$

I) The initial solution - the architectural independent and transparent specification (Fig.4.1)

Program FW1

initially $<\|\ t,u :: d[t,u] = w[t,u]>$

assign

$<[]\ t,u,v : 1 \le t,u,v < n :: d[t,u] := \min(d[t,u], d[t,v]+d[v,u])>$

end

Figure 4.1 Floyd-Warshall architecture independent

A model of the program is a weighted, directed graph, G, with n nodes. The weight of the edge connecting nodes t and u is denoted by w[t,u]. If nodes become disconnected, $w[t,u]=\infty$.

We address the issue of correctness by expressing the safety and the progress properties and by verifying them against the initial specification.

invariant (d[t,u] is the shortest path from node t to node u) holds for program FW1 (I).

II) The transformations of the initial solution based on causal and temporal order. We transform a union into a functionally equivalent composition of communication closed layers. The quantified layered representation is given by program FW2 (Fig.4.2). Next we explain why this solution can be more efficiently mapped to an asynchronous shared memory architecture or to a static distributed architecture. First we need to define some notations.

The initial communication closed layer is $L_0 = <[]\ t,u : 1 \le t,u \le n :: d[t,u] = w[t,u]>$.

The assignment d[t,u]=w[t,u] is denoted by $Q_{t,u}$, thus:$L_0 = <[]\ t,u : 1 \le t,u \le n :: Q_{t,u} >$.

One initializes the shortest distance between any nodes, t and u.

For any $1 \le i \le n$, $L_i = <[]\ t,u : 1 \le t,u < n :: d[t,u]: =\min(d[t,u], d[t,i]+d[i,u])>$ is a communication closed layer, which calculates the shortest distance between any nodes t and u, with intermediate node i. With the notation $P_{t,u}^{i}$ for the assignment d[t,u]: =min(d[t,u], d[t,i]+d[i,u]), we obtain:

$L_i = <[]\ t,u : 1 \le t,u < n :: P_{t,u}^{i}>$, where $1 \le i \le n$. Applying Lemma 2, the specification is rewritten as a communication closed layer composition:

```
Program FW2
assign
<[]t,u :: d[t,u] = w[t,u]>
• <• v :: 1 ≤v<n ::
      <[] t,u : 1 ≤t,u <n:: d[t,u]: =min(d[t,u], d[t,v]+d[v,u])>
  >
end
```

Figure 4.2 CCL solution FW2

$FW2 = L_0 \bullet L_1 \bullet \ldots \bullet L_n$. I_i = *invariant* (d[t,u] is the shortest path from node t to node u, with intermediate node i) - is an assertion about the global virtual state associated with the communication closed layer L_i. Applying the Lemma 4 and transitivity, the formula: "I_{k-1} *leads-to* I_k" holds for any $1 \le k \le n$.

III) Transformations of the layered solution to match static distributed architectures, by preserving safety and progress properties. We start with the layered representation.

$FW2 = L_0 \bullet L_1 \bullet \ldots \bullet L_n$, this is equivalent with the following specification:

$FW2 = <[]\ t,u : 1 \le t,u \le n :: Q_{t,u} > \bullet <[]\ t,u : 1 \le t,u \le n :: P_{t,u}^{1}> \bullet .. \bullet <[]\ t,u : 1 \le t,u \le n :: P_{t,u}^{n}>$

By explicitly representing the quantified expressions we obtain:

$FW2 = (Q_{1,1}\ []\ Q_{1,2}\ []\ \ldots\ []\ Q_{n,n}) \bullet (P_{1,1}^{1}\ []\ P_{1,2}^{1}\ []\ \ldots\ []\ P_{n,n}^{1}) \bullet \quad \bullet (P_{1,1}^{n}\ []\ P_{1,2}^{n}\ []\ ..\ []\ P_{n,n}^{n})$

The previous form applies to asynchronous shared memory architectures. We can take advantage of the minimal communication conflict closed ordering. Thus, if we apply the general algebraic laws of CCL to the program FW2 we obtain:

$FW2 = (Q_{1,1} \bullet P_{1,1}^{1} \bullet \ldots \bullet P_{1,1}^{n})\ []\ (Q_{1,2} \bullet P_{1,2}^{1} \bullet \ldots \bullet P_{1,2}^{n})\ [] \quad []\ (Q_{n,n} \bullet P_{n,n}^{1} \bullet .. \bullet P_{n,n}^{n})$

The second form is more appropriate for implementation on distributed architectures.

IV) The Meta-Program – a coordinated composition of the specification (FW2) with a mapping program (MP_{FW2}) of a mobile distributed architecture:

$MP_{FW2} = M_{FW2} \bullet FW2$

Reconfiguration of the moving elements typifies the distributed mobile architecture. The mapping program (M_{FW2}) stipulates a number of details related to mobility: location dependent computations, disconnected communication. The mapping program addresses initially only the issues of transient channel and disconnected operations. Later, we add gradually more details to represent mobility.

$M_{FW2} = (M_{1,1}\ []\ M_{1,2}\ []\ \ldots\ []\ M_{n,n})$

The notation $M_{t,u}$ stands for mapping program component that represents the events on the transient communication channel c_{tu} , connecting the mobile nodes t and u. If channel c_{tu} becomes disconnected, then the value of the variable d[t,u], the shortest distance between t and u, is to be replicated on the fly. As soon as the channel c_{tu} is reconfigured, then the value of the variable d[t,u] is updated (Fig. 4.3).

The Meta-Program representation is given by the communication closed layered composition of the mapping program with the specification FW2, provided in a form that matches a distributed architecture.

$$MP_{\mathrm{FW2}} = (M_{1,1} \;[]\; M_{1,2} \;[]\; .. \;[]\; M_{n,n}) \bullet (L_0 \bullet L_1 \bullet \ldots \bullet L_n)$$

$$MP_{\mathrm{FW2}} = (M_{1,1} \;[]\; M_{1,2} \;[]\; .. \;[]\; M_{n,n}) \bullet$$

$$(\, (Q_{1,1} \bullet P_{1,1}^{\;1} \bullet P_{1,1}^{\;2} \bullet ...\bullet P_{1,1}^{\;n}) \;[]\; (Q_{1,2} \bullet P_{1,2}^{\;1} \bullet P_{1,2}^{\;2} \bullet .. \bullet P_{1,2}^{\;n}) \;[]\; (Q_{n,n} \bullet P_{n,n}^{\;1} \bullet P_{n,n}^{\;2} \bullet ...\bullet P_{n,n}^{\;n})\,)$$

By applying the generalized laws of CCL on the Meta-Program MP_{FW2} we obtain:

$$MP_{\mathrm{FW2}} = (M_{1,1} \bullet (Q_{1,1} \bullet P_{1,1}^{\;1} \bullet P_{1,1}^{\;2} \bullet ...\bullet P_{1,1}^{\;n})) \;[]$$

$$(M_{1,2} \bullet (Q_{1,2} \bullet P_{1,2}^{\;1} \bullet P_{1,2}^{\;2} \bullet ...\bullet P_{1,2}^{\;n})) \;[]\; \ldots \;[]$$

$$(M_{n,n} \bullet (Q_{n,n} \bullet P_{n,n}^{\;1} \bullet P_{n,n}^{\;2} \bullet ...\bullet P_{n,n}^{\;n}))$$

We distribute the mapping component on all program specification components. The final form, a quantified representation, is as follows:

$$MP_{\mathrm{FW2}}=<[]t,u:\ 1 \le t,u \le n :: (M_{t,u} \bullet Q_{t,u}) \bullet <\bullet\ k :: M_{t,u} \bullet P_{t,u}^{\;k}>>$$

The Meta-program MP_{FW2} is a causal coordinated composition of mapping and specification events. Specification is transparent and architecture-independent. Mapping deals with mobility's design details. This separation of concerns allows for reusing the design knowledge of static distributed algorithms in Meta-Programs, and is the basis for a reduced design cost and less complexity.

Program $M_{t,u}$

declare

connected(c_{tu}), disconnected(c_{tu}) : *adapter*

reconfigured(c_{tu}): *adapter*

assign

d[t,u]:=new(d[t,u]) *self-adapt* connected(c_{tu}),

~ replicate(d[t,u]) *self-adapt* disconnected(c_{tu}),

~ update(d[t,u]) *self-adapt* connected(c_{tu}),

end

Figure 4.3 A Mapping Program Component

V) The verification of the coordinated *Meta-Program* . We can infer the safety and progress properties in the Meta-Program MP_{FW2} by causally composing mapping and specification data and temporal properties and by applying Lemma 4, Lemma 5 and the generalized algebraic laws of CCL.

5 CONCLUSIONS

This paper describes DUALITY, a design methodology for a systematic specification and verification of mobile distributed systems. We integrate the simplicity of UNITY's notation and logic with the modular structure of the communication-closed layers. In this way, we describe, in a more abstract manner, the causality of events in a mobile distributed program. Our definition of Meta-Program extends the UNITY's notion of program-schema. The Meta-Program reflects the duality of concurrency and causality relations between events in a mobile distributed system. The strength of our design model is the capacity to synchronize, to coordinate specifications and mappings in a unifying *Meta-Program*. DUALITY allows the reuse of architectural patterns and proofs of static distributed algorithms in the design of mobile distributed programs.

The correctness of mobile distributed programs should not rely on implementation details related to a certain target machine or concerning issues of some particular mobile architecture. By introducing DUALITY, the correctness of mobile distributed systems is addressed in a systematic manner with each transformation of the specification. Applicable for initial phases of the design, DUALITY principles supports a simple formalism that, in the end, can be mapped onto real programming languages. This approach leads to fewer, but more relevant design alternatives, therefore less design complexity.

We are currently working on to extend the formalism of communication-closed layer composition, defined in terms of UNITY and branching temporal logic. Ongoing work integrates a partial-order model checker [7] to provide a mechanized verification for specifications and mappings of mobile distributed programs.

REFERENCES

[1] Amadio, R., "An asynchronous model of locality, failure and process mobility", Coordination Languages and Models, Springer-Verlag, Berlin, 374-391, 1997.

[2] Emerson, A.,"Temporal and Modal Logic", *Handbook of Theoretical Computer Science*, Elsevier Science Publisher B.V., 1990.

[3] Chandy, K., Misra, J., *Parallel Programs Design: A Foundation*, Addison-Wesley, New-York, N.Y., 1988.

[4] Elrad, T., Francez, N., "Decomposition of Distributed Programs into Communication- Closed Layers", *Science of Programming*, 2, 155-173, N.H., 1982.

[5] T. Elrad, K, Nambi, "Scheduling Cooperative Work: Viewing Distributed Systems as Both CSP and SCL", *In Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh May, 1993.

[6] Francez, N, *Fairness*, Springer-Verlag, New York, N.Y, 1986.

[7] Godefroid, P., Peled, D., Staskauskas, M., "Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs", IEEE Transactions on Software Engineering, 2:7, 496-507,1996.

[8] Janssen, W., Poel, M, Zwiers, J., "Actions system and action refinement in the development of parallel systems", *In Proc. of CONCUR'91*, 298-316, Springer-Verlag, LNCS 527, 1991.

[9] McCann, P.J.,Roman, G.C., Plun,J.Y., "Mobile-UNITY: Reasoning and specification in mobile computing," ACM Transactions on Software engineering and Methodology, 6:3,250-282,1997.

[10] Misra, Jayadev, "A logic for concurrent programming: Progress", *Journal of Computer and Software Engineering*, 3:2, 273-300, 1995.

[11] Misra, Jayadev, "A logic for concurrent programming: Safety", *Journal of Computer and Software Engineering*, 3:2, 239-272, 1995.

[12] Katz, S., Peled, D., "Interleaving Set Temporal Logic", *Theoretical Computer Science*, 75(2), 1990.

[13] Poel, M, Zwiers, J, "Layering Techniques for Development of Parallel Systems", *Proc. of ACM PODC*, 1992.

[14] Sanders, B., Meyer, D., "Composing Lead-to Properties", Technical Report, UF-CISE, ftp://ftp.cise.ufl.edu/cis/tech-reports/tr96/tr96-013.ps

[15] Stomp, F.A., de Roever, W.P., "Designing distributed algorithms by means of formally phased reasoning.", *Proc. of the 3rd International Workshop on Distributed Algorithms*, LNCS 392, 242-253, Springer-Verlag, 1989

[16] Badrinath, B. R., Welling, G., "Event Delivery Abstractions for Mobile Computing", Rutgers University Technical Report LCSR-TR-242, 1997.