Petri nets for the verification of Ubiquitous Systems with Transient Secure Association[⋆]

Fernando Rosa-Velardo

Technical Report 2/07
Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid
{fernandorosa}@sip.ucm.es

Abstract. *Transient Secure Association* has been widely accepted as a possible alternative to traditional authentication in the context of Ubiquitous Computing. In this paper we develop a formal model for the *Resurrecting Duckling Policy* that implements it. Our model, that we call TSA systems, is based on Petri Nets, thus obtaining an amenable graphical representation of our systems. We prove that TSA specifications have the same expressive power as P/T nets, so that coverability, that can be used to specify security properties in this setting, is decidable for TSA systems. Then we address the problem of implementing TSA systems with a lower level model that only relies on the secure exchange of secret keys. We prove that if we view these systems as closed systems then our implementation is still equivalent to P/T nets. However, if we consider an open framework then we need a mechanism of fresh name creation to get a correct implementation. This last model is not equivalent to P/T nets, but the coverability problem is still decidable for them, even in an open setting, so that checking the security properties of the represented systems remains decidable.

1 Introduction

The term *Ubiquitous Computing* was coined by Mark Weiser [21] to describe environments full of devices that compute and communicate with its surrounding context and, furthermore, interact with it in a highly distributed but pervasive way [20]. This computing paradigm gives rise to a great deal of new situations that produce new problems, in the context of mobility, coordination and security, among others. For a survey of the challenges posed by Ubiquitous Computing see [10].

As mentioned above, the new framework of ubiquitous computing affects traditional security assumptions. Authentication is probably the major security problem for ubiquitous systems, since it is a precondition for other properties

[⋆] Work partially supported by the Spanish projects MIDAS TIC 2003-01000, MASTER TIC 2003-07848-C02-01 and PROMESAS-CAM S-0505/TIC/0407.

as confidentiality or integrity. But in this new context, we can no longer rely on an always online policy, due to the unreliable nature of ad hoc nets. Thus, we cannot approach the problem of authentication by assuming the existence of an online server that verifies in real time the (global) identity of principals, nor the existence of public key repositories.

In order to solve this problem, several weak forms of authentication that substitute global identity-based authentication have been proposed. One of the proposals is that of *Transient Secure Association*, implemented by the *Resurrecting Duckling Policy* [18]. This policy is based on the fact that sometimes a principal may not know anything at all about some concrete device (e.g., when a user has just bought a PDA), but still it wants to be the only one to use it (e.g., because she is the first one who gets to push the button). In this case, even if they have no prior knowledge of each other, they can still become associated, so that from then on they share an asymmetric relation, one being a slave and the other becoming its master. Other proposals are based on the notion of *trust* [2], though we will not consider them here.

Another issue of great importance in the field of ubiquitous computing is that of Coordination. Indeed, inherent to ubiquitous computing is the continuous change of state of the different components that form a system, that need to coordinate to achieve their goals. This fact, together with the unreliable nature of ad hoc nets and the heterogeneity of the components make the orchestration approach unrealistic and little robust, so that a choreographic coordination is more advisable in this setting. Therefore, a key technology is service discovery, which nowadays is based on a heavy standardization [12] (as in Sun's Jini or in Microsoft's UPnP).

In this paper we develop a formal model for choreographic coordination in ubiquitous systems, with primitives implementing the resurrecting duckling policy. The model, called TSA, is based on Petri nets [3]. By using Petri Nets we obtain an amenable graphical representation of our systems, together with a number of theoretical results for their analysis, which would not hold in other more expressive models. In particular, we will prove that TSA specifications can be seen as a syntactic sugar formalism for P/T nets. Moreover, we will see that we can implement TSA systems by means of Mobile Synchronizing Petri Nets, which can be seen as a class of Coloured Petri Nets [8] with a single colour of pure names [6], and syntactic sugar for mobility, except for the fact that they can create fresh names, in a way borrowed from π-calculus [9]. We have proved several useful decidability results for them, which are inherited by TSA systems, even though TSA systems may represent infinite state systems. In particular, decidability of coverability in MSPN system implies that all the properties of TSA systems which can be stated in terms of coverability are decidable.

The remainder of the paper is organized as follows. Section 2 gives an overview of the Resurrecting Duckling policy. In Section 3 we define our model for Transient Secure Association. Section 4 presents a simple example. In Section 5 we prove the simulation results and, finally, in Section 6 we present our conclusions and directions for further work.

$$\begin{tikzcd} \mathit{dead} \arrow[rr, bend left, "\text{imprint}"] & & \mathit{alive} \arrow[ll, bend left, "\text{kill}"] \end{tikzcd}$$

Fig. 1. The resurrecting duckling policy

2 The Resurrecting Duckling policy

Let us briefly describe Transient Secure association and the Resurrecting Duckling security policy model. Transient secure association is about creating a link between two components, that establishes a master-slave relationship (typically between a controller and a peripheral). This link needs to be secure, so that no other principal can play the role of either the master or the slave. This produces a tree topology of master-slave relationships between components. The association must also be transient, in the sense that the topology can change if desired (by the master) along the execution of the system.

The Resurrecting Duckling policy model implements transient secure association with the following metaphor: *A duckling that emerges from its egg recognizes as its mother the first moving object that emits a sound.* From then on, the duckling obeys its mother duck until its death. Similarly, a peripheral can recognize as its mother duck the first entity that sends it a secret key, called *imprinting key*, which it will use to recognize its master from then on. However, in order to make this association transient, we assume that the duckling can die and resurrect, and then it can recognize as its mother duck a different entity, to which it will be faithful from that point on.

The policy can be described as a list of four principles:

1. **Two State principle.** Entities can be either *imprintable* (dead) or *imprinted* (alive). Imprintable entities can be imprinted by anyone and imprinted entities only obey their mother duck.
2. **Imprinting principle.** The step from imprintable to imprinted is called *imprinting*, and happens when an entity, from now on the mother duck, sends a key to the duckling (see Fig. 1).
3. **Death principle.** The step from imprinted to imprintable is called *death* (see Fig. 1). The default cause of death is the mother duck's order to its duckling to commit *seppuku*[1], though other causes are envisaged, as death by old age or by completion of a task.
4. **Assassination principle.** It must be uneconomical for an attacker to kill a duckling (to cause its death).

[1] Traditional ritual suicide from Japanese samurai

3 TSA systems

Our aim is to define a model for ubiquitous systems, that we call TSA, based on Petri Nets and with primitives implementing the Resurrecting Duckling policy. It will be a variation of the one developed in [5]. A TSA system is essentially a set of localized nets that can move between locations and can imprint other nets, give orders to the nets they have previously imprinted, kill those nets, and being imprinted or receive orders from its master when imprinted. In order to define our model we have to specify some of the aspects of the policy that were not still completely precise.

We assume that the different components of a system have a type, taken from a set $\mathcal{O}$. We use a special type $\top \in \mathcal{O}$ for the type of components that need not be imprinted to perform their potential behaviour (e.g., users). Components can be dead or alive. Top or imprinted components are alive, otherwise they are dead. Those alive components can imprint dead ones, thus creating an unique and asymmetrical link between them. By asymmetrical we mean that from then on the imprinted component becomes a slave of the other, doing everything its master orders to it. However, when executing its code, the slave component can as well imprint other components, thus creating a hierarchical relation between components. Therefore, not only the component being killed becomes death, but also all of its slaves and so on, recursively.

We assume that components can only be imprinted within a certain range, that depends on the imprinting method being used (e.g., physical contact [17]). We abstract from those particular methods and simply assume that the imprinting can take place whenever both components are in the same location. In particular, we assume that the key exchange that takes place when imprinting a component is done in a secure way. Besides, a component can only give orders to those slave components that are currently co-located with it.

Dually, a slave component can be killed by its owner only when they are co-located. Thus, we are implementing the default variant of the death principle, in which the death of the duckling is caused by the mother duck. Finally, we assume that the principals can only interact with each other using the model's primitives, so that the assassination principle of the policy also holds, since attackers cannot kill any duckling in a way not established by the policy.

We will use the set $\mathcal{L}$, a set of location names. $\mathcal{O}$ is a set of component types, with a special element $\top \in \mathcal{O}$. We use a set S for the syntactic primitives of the standard used for coordination and $\mathcal{A}$ to denote autonomous actions. $\mathcal{A}$ contains among others the labels *go* k for every $k \in \mathcal{L}$. We denote $S? = \{s? \mid s \in S\}$ and $S! = \{s! \mid s \in S\}$. For $m > 0$, we denote by $\mathit{Labels}(m)$ the set $\mathcal{A} \cup S? \cup \bigcup_{i=1}^{m} (\{i\} \times \mathit{Labels}_{\mathit{Auth}})$, where $\mathit{Labels}_{\mathit{Auth}} = S! \cup \{\mathit{imprint}(o) \mid o \in \mathcal{O} \setminus \{\top\}\} \cup \{\mathit{kill}\}$.

Definition 1. *A TSA component is a tuple $N = (P_N, T_N, F_N, \lambda_N, o_N)$ such that P_N and T_N are disjoint finite set of places and transitions, respectively, $F_N \subseteq (P_N \times T_N) \cup (T_N \times P_N)$, $\lambda_N : T_N \to \mathit{Labels}(m)$ for some $m > 0$ and*

$o_N \in \mathcal{O}$. We say that N has type o_N, that m is the capacity of N, and we denote by $cap(N)$ the set $\{1, \ldots, m\}$.

A TSA component is a typed-labelled Petri net. The capacity of a component represents the number of slaves it can have simultaneously imprinted. Its type o is just a syntactic category, which can be intuitively understood as the type of device it is (e.g., PDA, electronic note,...), as the imprinting protocol it uses when being imprinted or both simultaneously. The labelling of the transitions of the net establishes a partition in its transition set: Those transitions t with $\lambda(t) \in \mathcal{A}$ are autonomous transitions, that is, those that can be executed in an autonomous way, without needing to synchronize with others; If $\lambda(t) \in S?$ the transition is used to receive orders from their masters, when imprinted; Otherwise, we have that $\lambda(t) \in \{i\} \times \mathit{Labels}(m)$. The set $\mathit{Labels}(m)$ is the disjoint union of m identical sets Labels_{Auth}, each of them used to *communicate* with a different i-slave. If some net fires t with $\lambda(t) = (i, \mathit{imprint}(o))$ then some component of type o is henceforth a i-slave of it. If $\lambda(t) = (i, s!)$ then the firing of t instructs the corresponding slave to perform action $s?$. Finally, if $\lambda(t) = (i, \mathit{kill})$ then its i-slave is instructed to commit seppuku. Next we introduce these concepts formally.

As usual, given a transition t we will denote by ${}^\bullet t = \{p \mid (p, t) \in F\}$, the set of preconditions of t and by $t^\bullet = \{p \mid (t, p) \in F\}$, its set of postconditions. If $cap(N) = \{1\}$ and $\lambda(t) = (1, \ell)$ we will simply write $\lambda(t) = \ell$.

Definition 2. *A TSA system $\mathcal{N}$ is a set of pairwise disjoint TSA components.*

Intuitively, the nets N with $o_N = \top$ are components that need not be imprinted in order to execute their actions. We will denote by $P_\mathcal{N}$ the set of all places in $\mathcal{N}$ and by $T_\mathcal{N}$ the set of all transitions in $\mathcal{N}$, or just P and T when there is no confusion.

Next we define the part of the state of the system regarding the dependency relations between its components.

Definition 3. *Given a TSA system $\mathcal{N}$, a dependency function of $\mathcal{N}$ is a partial mapping $R : \{(N, j) \mid N \in \mathcal{N}, j \in cap(N)\} \to \mathcal{N}$. A dependency function R induces a dependency graph $G(R)$ with set of nodes $\mathcal{N}$ and an arc from N to N' labelled by j if $R(N, j) = N'$, in which case we say that N' is a j-slave (or just a slave) of N in R.*

We will call *directed tree* to any directed graph in which there is a node, that we call *root* of the tree, so that for every node in the graph there is a single path from the root to it. Therefore, a *directed forest* is a set of directed trees. Given two nodes s and s' of a directed forest, we will say s' is a descendant of s if there is a path in the forest from s to s'. Moreover, given a set A we will denote by $\mathcal{MS}(A)$ the set of multisets with elements in A.

Definition 4. *A marking of a TSA $\mathcal{N}$ is a tuple $\mathcal{M} = (M, loc, R)$, where $M \in \mathcal{MS}(P_\mathcal{N})$, $loc : \mathcal{N} \to \mathcal{L}$ and R is a dependency function of $\mathcal{N}$ such that:*

- *$G(R)$ is a directed forest,*
- *For every $N \in \mathcal{N}$, if $o_N = \top$ then N is not a slave in R.*

Therefore, a marking is composed by an ordinary marking for Petri nets, a function specifying the locality in which each component is present, and a dependency function.

We will say a component N is alive in R if $o_N = \top$ or it is a slave in R. Otherwise, we will say it is dead. The components N and N' are co-located if $loc(N) = loc(N')$.

Now let us define the behaviour of TSA systems. In general, only alive components can fire transitions. Next we define the firing of autonomous transitions, that are as ordinary transitions in P/T nets, that remove tokens from precondition and add them in postconditions, except that they are restricted to alive components and can cause the movement of the executing net. We will denote by $+$ and $-$ the multiset union and the multiset difference, respectively, to distinguish them from $\cup$ and $\setminus$, the corresponding operations over sets.

Definition 5. *Let $\mathcal{N}$ be a TSA system, $\mathcal{M} = (M, loc, R)$ a marking of $\mathcal{N}$, N an alive component in R, and $t \in T_N$ such that $\lambda(t) \in \mathcal{A}$. We say that t is enabled in $\mathcal{M}$ if $^\bullet t \subseteq M$. In that case we say that t can be fired and then the marking (M', loc', R) is reached, where:*

- *$M' = M - {^\bullet t} + t^\bullet$.*
- *If $\lambda(t) = go\ k$ then $loc'(N) = k$ and $loc'(N) = loc(N)$ for every $N' \neq N$. Otherwise, $loc' = loc$.*

Now we define the firing of imprinting transitions. The net firing any of them specifies the type it wants to imprint.

Definition 6. *Let $\mathcal{N}$ be a TSA system, $\mathcal{M} = (M, loc, R)$ a marking of $\mathcal{N}$, N and N' an alive and a dead component in R, respectively, both co-located, and $t \in T_N$ such that $\lambda(t) = (i, imprint(o_{N'}))$. We say that t is enabled in $\mathcal{M}$ if*

- *$^\bullet t \subseteq M$*
- *$(N, i) \notin Dom(R)$*

Then t can be fired, to reach the marking (M', loc, R'), where $M' = M - {^\bullet t} + t^\bullet$ and R' extends R with $R'(N, i) = N'$.

Therefore, the imprinted net must be dead in order to be imprinted, and of the type required by the imprinting component. Moreover, the imprinting net must not already have an i-slave net.

Let us now define the transitions that represent orders from masters to slaves.

Definition 7. *Let $\mathcal{N}$ be a TSA system, $\mathcal{M} = (M, loc, R)$ a marking of $\mathcal{N}$, N and N' two co-located alive components such that N' is a i-slave of N in R. Let us also consider $t \in T_N$ such $\lambda(t) = (i, s!)$ and $t' \in T_{N'}$ such that $\lambda(t') = s?$. We say that the pair of transitions (t, t') is enabled in $\mathcal{M}$ if $^\bullet t \cup {^\bullet t'} \subseteq M$. Then the pair (t, t') can be fired, getting (M', loc, R) where $M' = M - {^\bullet t} - {^\bullet t'} + t^\bullet + t'^\bullet$.*

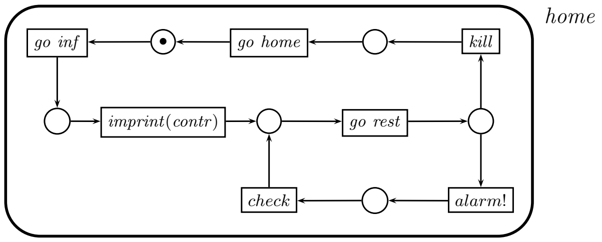

Fig. 2. Nurse

Therefore, if N' is an i-slave of N then N can give orders to N', which is formalized by a synchronous firing of two transitions. Finally, let us define how a master can kill one of its slaves.

Definition 8. *Let $\mathcal{N}$ be a TSA system, $\mathcal{M} = (M, loc, R)$ a marking of $\mathcal{N}$, N and N' two co-located alive components such that N' is a i-slave of N, and $t \in T_N$ such $\lambda(t) = (i, kill)$. We say that t is enabled in $\mathcal{M}$ if ${}^\bullet t \subseteq M$. Then t can be fired, getting (M', loc, R') where $M' = M - {}^\bullet t + t^\bullet$ and R' is the result of*

- *removing (N, i) from the domain of R, and*
- *removing $(\overline{N}, j)$ from the domain of R, for every $\overline{N}$ descendant of N' in $G(R)$ and every j.*

Therefore, all the components that depend of N' are also indirectly killed. Notice that this is so even if those components are not co-located with N and N'. Alternatively, we could force N' to explicitly search and kill its slaves, and so on, but this would not modify the potential behaviour of any of the so killed components. This is so because these nets can perform two kind of actions: autonomous actions, that could have been performed anyway before their death; and synchronizations with its master, which will not happen after the seppuku order.

We will use $u, u', \ldots$ to range over transitions or pairs of transitions. Thus, in any of the previous cases we will write $\mathcal{M}[u\rangle\mathcal{M}'$ if $\mathcal{M}'$ is the result of firing u in $\mathcal{M}$.

Since alive components cannot be imprinted, and only alive components can imprint, whenever the dependency graph is initially a directed forest, it remains so. Therefore, we have the following result.

Proposition 1. *If $\mathcal{M}$ is a marking of a TSA system $\mathcal{N}$ and $\mathcal{M}[u\rangle\mathcal{M}'$ then $\mathcal{M}'$ is also a marking of $\mathcal{N}$.*

Reachability for TSA systems is defined as expected. We say a marking (M, loc, R) can be covered if there is a reachable marking (M', loc', R') such that $M \subseteq M'$, $loc = loc'$, $Dom(R) \subseteq Dom(R')$ and for every $(N, i) \in Dom(R)$, $R(N, i) = R'(N, i)$. The coverability problem consists on deciding if a given marking can be covered.

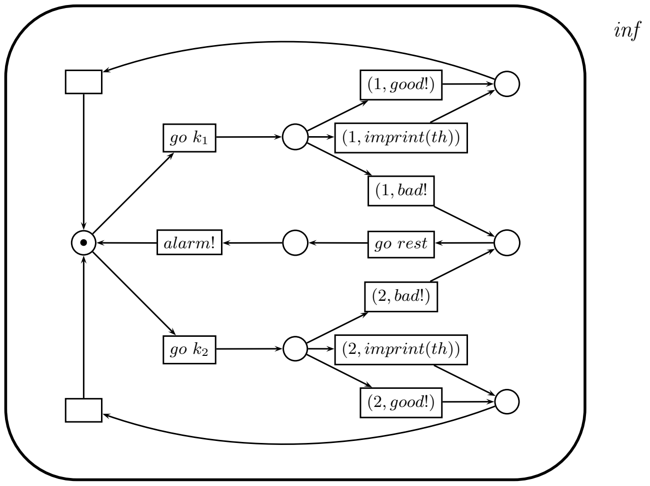

Fig. 3. Thermometer controller

4 An application example

In this section we present a simple application example to illustrate the definitions in the previous section. Let us consider a scenario with several patients in two rooms of a hospital, called k_1 and k_2. Each of the patients has an electronic thermometer [19] that reads the temperature every once in a while, and willing to send (e.g., using RFID) a message stating whether everything is all right or not. We assume that these thermometers are inactive (that is, dead according to our terminology), except when a thermometer controller takes over them. Intuitively, we assume that every room has one of these controllers, that reads the messages output by thermometers, and sends an alarm message to a nurse whenever the temperature is not correct. For simplicity, we assume that all these controllers are in fact part of a single component that moves between rooms in the hospital and that it can only simultaneously control two thermometers, so that it has capacity two.

Finally, there is a night shift nurse, that must check the temperature of the patients. For that purpose, it may imprint a controller, so that she is the only one to receive alarm messages from it. Therefore, the life cycle of a night-shift nurse consists on going from her home to the infirmary, imprinting a thermometer controller, going to rest, checking the patient in case she receives an alarm message, or killing the controller and going back home.

We model this scenario by means of a TSA system with several components. The first one, shown in Fig. 2, represents the nurse, that we assume has type $\top$. Initially, it is located in location *home*. Then it can fire *go inf*, thus moving to the infirmary, where it can imprint the thermometer controller and then go to rest. Notice that, since the nurse has capacity one, we use simple names as labels of its synchronizing transitions, such as *kill*, instead of $(1, \mathit{kill})$.

$$\begin{tikzcd}
 & \mathit{nurse} \arrow[d] & \\
 & \mathit{controller} \arrow[dl, "1"'] \arrow[dr, "2"] & \\
\mathit{thermometer} & & \mathit{thermometer}
\end{tikzcd}$$

Fig. 4. Dependency relation

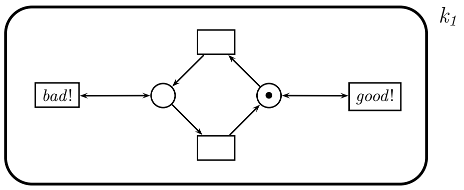

Fig. 5. Patients

The thermometer controller, with type *contr*, is shown in Fig. 3, initially in location *inf*. The controller, however, does not have a $\top$ type, so that it must wait to be imprinted to perform any action. After been imprinted, it can go either to location k_1 or to k_2. In these locations there are patients, all of them like that in Fig. 5. Now the controller has a choice between *good*!, *bad*! or *imprint*(*th*). However, since the patient thermometers are still dead (in our sense), all the controller can do at the present time is imprinting them. Then it can again choose between any of the locations. If it goes back again to the same place then it can read the thermometer and, if it reads *bad*, that is, if it synchronizes with the thermometer in transition *bad*!, then it goes to the rest room and sends an alarm message to the nurse.

Eventually, the system will reach a dependency state like the one shown in Fig. 4, with both thermometers imprinted by the controller. At any point, the nurse can fire its *kill* transition, thus killing the controller (and, recursively, the controller killing its thermometers). Notice that the controller does not have any killing transition, so that it only kills its thermometers when, in turn, the nurse kills it.

5 Verification of TSA systems

In the previous sections we have presented a model that allows us to specify systems using primitives that implement the resurrecting duckling policy, which are correct by definition. In this section we will see how we can use the known decidability results for the verification of the obtained TSA systems.

First of all, we will prove that the locality component of TSAs is not essential in the model. More precisely, we will say a TSA system is centralized if none of its components have movement transitions and all of them are initially located in the same location. Under these assumptions, we can ignore the locality component of centralized TSA systems, thus writing just (M, R) instead of (M, loc, R). In the following result we will write $\mathcal{A}_c$ to denote the set $\mathcal{A} \setminus \{go\ k \mid k \in \mathcal{L}\}$.

Lemma 1. *Every TSA system can be simulated by a centralized TSA system.*

Proof. Let $\mathcal{N}$ be a TSA system. Given $N \in \mathcal{N}$ we will define the centralized TSA component N^*, so that the centralized system $\mathcal{N}^* = \{N^* \mid N \in \mathcal{N}\}$ simulates $\mathcal{N}$. Let $\mathcal{K} = \{k \mid \exists t \in T_{\mathcal{N}}, \lambda(t) = go\ k\} \cup loc_0(\mathcal{N}) \subseteq \mathcal{L}$, which is the set of localities in which components can reside in any reachable marking. Since that set is finite, we can add for each component N one place $N@k$ per locality $k \in \mathcal{K}$, all marked in mutual exclusion, thus representing the current locality of each component as part of its ordinary marking. Then, for each non-autonomous transition we will consider also one per locality, so that a transition t being fired in k is represented by the firing of a transition $t(k)$. The place $N@k$ must be a pre/postcondition of transition $t(k)$, and movement transitions must change the tokens in the new places accordingly. Then, if $N = (P, T, F, \lambda, o)$ we take $N^* = (P^*, T^*, F^*, \lambda^*, o)$, where

– $P^* = P \cup \{N@k \mid k \in \mathcal{K}\}$
– $T^* = \{t \in T \mid \lambda(t) \in \mathcal{A}_c\} \cup \{t(k) \mid \lambda(t) \notin \mathcal{A}_c,\ k \in \mathcal{K}\}$
– $F^* = \{(p,t) \in F \mid \lambda(t) \in \mathcal{A}_c\} \cup \{(t,p) \in F \mid \lambda(t) \in \mathcal{A}_c\} \cup \{(p, t(k)) \mid (p,t) \in F,\ \lambda(t) \notin \mathcal{A}_c\} \cup \{(t(k), p) \mid (t,p) \in F,\ \lambda(t) \notin \mathcal{A}_c\} \cup \{(N@k, t(k)), (t(k), N@k) \mid k \in \mathcal{K},\ \lambda(t) \neq go\ k'\} \cup \{(t(k), N@k') \mid \lambda(t) = go\ k'\}$
– $\lambda^*(t) = \lambda(t)$ and $\lambda^*(t(k)) = \lambda(t)$

We assume that a transition in the simulating centralized system labelled by $go\ k$ is just an autonomous transition. For a given reachable marking $\mathcal{M} = (M, loc, R)$ we define $\mathcal{M}^* = (M^*, R)$, where $M^*(p) = M(p)$ for every $p \in P$, $M^*(N@loc(N)) = 1$ and $M^*(N@k) = 0$ if $k \neq loc(N)$ for every N. Then, it holds that

$$\mathcal{M}_1[u\rangle\mathcal{M}_2 \Leftrightarrow \mathcal{M}_1^*[u^*\rangle\mathcal{M}_2^*$$

where, if $\mathcal{M}_1 = (M_1, loc_1, R_1)$ then

$$u^* = \begin{cases} u \text{ if } \lambda(u) \in \mathcal{A}_c \\ u(k) \text{ if } \lambda(u) = go\ k' \\ (t(k), t'(k)) \text{ if } u = (t, t') \in T_N \times T_{N'},\ loc_1(N) = loc_1(N') = k \end{cases}$$

Therefore, in the following we can assume that every TSA system is centralized. Now, we prove that every TSA system can be simulated by a P/T net.

Proposition 2. *Every TSA system can be simulated by a P/T net.*

Proof. According to the previous result we can assume that we are given a centralized TSA system $\mathcal{N}$ with initial marking $\mathcal{M}_0 = (M_0, R_0)$, and we must construct the P/T net $N^* = (P^*, T^*, F^*, M_0)$ that simulates it. Let us first introduce some notations. We write $OK(t, t', R)$ whenever $t \in T_N$, $t' \in T_{N'}$ and both N and N' are alive and N' is a slave of N, according to R. Analogously, we write $OK(t, N, R)$ whenever $\lambda(t) = imprint(o_N)$ and N is dead according to

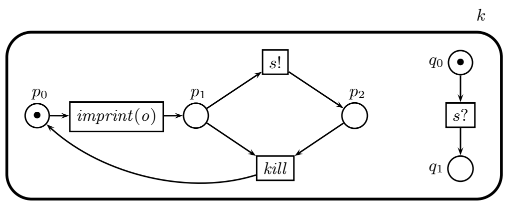

Fig. 6. Simple example of TSA system

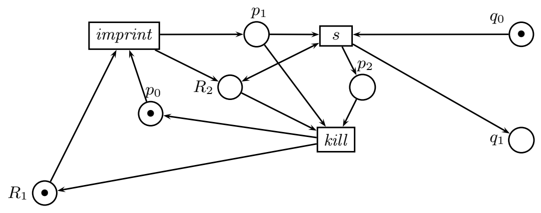

Fig. 7. Translation of the TSA system in Fig. 6

R, or whenever $\lambda(t) = kill$ and N is alive according to R. Finally, given a dependency function R, we will write $R(t, N)$ to denote the dependency function that results when firing t with N as goal. More precisely, if $\lambda(t) = (i, imprint(o_N))$ then $R(t, N)$ is the result of adding to R, $R(N', i) = N$, where $t \in T_{N'}$. Analogously, if $\lambda(t) = (i, kill)$ then $R(t, N)$ is the result of removing (i, N') from the domain of R, as well as every descendant of N.

In order to simulate a TSA system by means of a P/T net, for each component N we add places $dead(N)$ and $alive(N)$, marked in mutual exclusion, thus stating whether that component is dead or alive. Since there are a finite number of components and each of them has a finite capacity, there is only a finite number of dependency functions. Then we can have one place for each one of them, also marked in mutual exclusion, and used to restrict synchronizations to happen only in the right configurations. For instance, for each pair (t, t') of synchronizing transitions labelled respectively by $s?$ and $(i, s!)$ we consider for every dependency function R the transition (t, t', R) that represents the synchronized firing of (t, t') in a dependency state R. Of course, this synchronization can only happen when the net firing t' is an i-slave of the one firing t' according to R. Therefore, this transition must have R as pre/postcondition, as well the preconditions and postconditions of both t and t'. An analogous construction applies to killing and imprinting transitions.

Then, let us take:

– $P^* = P_{\mathcal{N}} \cup \{alive(N), dead(N) \mid N \in \mathcal{N}\} \cup \{R \mid R \textit{ dependency function of } \mathcal{N}\}$
– $T^* = \{t \in T_{\mathcal{N}} \mid \lambda(t) \in \mathcal{A}\} \cup$
  $\{(t, t', R) \mid \lambda(t) = s?, \lambda(t') = (i, s!), OK(t, t', R)\} \cup$
  $\{(t, N, R) \mid OK(t, N, R)\lambda(t) = kill \textit{ or } \lambda(t) = imprint(o_N)\}$
– $F^* = F_1 \cup F_2 \cup F_3$, where
  • $F_1 = \{(p, t) \mid \lambda(t) \in \mathcal{A},\ p \in {}^\bullet t\} \cup \{(t, p) \mid \lambda(t) \in \mathcal{A},\ p \in t^\bullet\} \cup \{(alive(N), t), (t, alive(N)) \mid t \in T_N\}$

- $F_2 = \{(p,(t,t',R)) \mid p \in {}^\bullet t \cup {}^\bullet t'\} \cup \{((t,t',R),p) \mid p \in t^\bullet \cup t'^\bullet\} \cup \{(R,(t,t',R)),((t,t',R),R) \mid (t,t',R) \in T^*\}$
- $F_3 = \{(p,(t,N,R)) \mid p \in {}^\bullet t\} \cup \{((t,N,R),p) \mid p \in t^\bullet\} \cup \{(R,(t,N,R)) \mid (t,N,R) \in T^*\} \cup \{((t,N,R),R(t,N))\} \cup \{(\mathit{alive}(N),(t,N,R)) \mid \lambda(t) = \mathit{kill}\} \cup \{((t,N,R),\mathit{dead}(N)) \mid \lambda(t) = \mathit{kill}\} \cup \{(\mathit{dead}(N),(t,N,R)) \mid \lambda(t) = \mathit{imprint}(o_N)\} \cup \{((t,N,R),\mathit{alive}(N)) \mid \lambda(t) = \mathit{imprint}(o_N)\}$

Finally, given a marking $\mathcal{M} = (M,R)$ of $\mathcal{N}$ we define the marking $\mathcal{M}^*$ of $\mathcal{N}^*$ as follows:

- $\mathcal{M}^*(p) = M(p)$ for every $p \in P_\mathcal{N}$
- $\mathcal{M}^*(R) = 1$ and $\mathcal{M}^*(R') = 0$ for every $R' \neq R$
- $\mathcal{N}^*(\mathit{alive}(N)) = \begin{cases} 1 \text{ if } N \text{ alive in } R \\ 0 \text{ otherwise} \end{cases}$
- $\mathcal{N}^*(\mathit{dead}(N)) = \begin{cases} 1 \text{ if } N \text{ dead in } R \\ 0 \text{ otherwise} \end{cases}$

Using these notations, it holds that

$$(M_1,R_1)[u\rangle(M_2,R_2) \Leftrightarrow (M_1,R_1)^*[u^*\rangle(M_2,R_2)^*$$

where $u^* = \begin{cases} t \text{ if } u \in T_\mathcal{N} \text{ and } \lambda(t) \in \mathcal{A} \\ (t,t',R_1) \text{ if } u = (t,t') \\ (t,N,R) \text{ if } u = t \text{ and } \lambda(t) = \mathit{kill} \text{ or } \lambda(t) = \mathit{imprint}(o_N) \end{cases}$

Figure 7 shows the P/T net that simulates the TSA system in Fig. 6, assuming that the net in the right has type o. In it we use place R_1 for the dependency function in which there is no associations between components, and R_2 for that in which the component in the left of Fig. 6 is master of the one in the right. Then, the transition labelled by *imprint* can only be fired when there is a token in R_1, and that firing has the effect of moving that token from R_1 to R_2, as well as moving the token in p_0 to p_1. The killing transition has an analogous behaviour. Finally, transitions $s!$ and $s?$ are merged into a single transition s, that has R_2 as a pre/postcondition place, because it can only happen when both components are associated, and it does not change this association. This construction, even for the simple example shown here, produces the typical spaghetti problem of P/T nets, though, fortunately, it can be carried out automatically.

By applying this result, we can use all the existing machinery for ordinary P/T nets in the analysis of our systems. For instance, coverability[2] and home state problems are known to be decidable for P/T nets (for a survey see [4]). Then, in principle, we could algorithmically check any property that can be stated in terms of them. For example, regarding the example in the previous section, we could check that, whenever a thermometer is imprinted by a doctor, then whenever needed it eventually raises the alarm, and only to the right doctor, since that is a home state property. Another property that could be interesting

[2] In fact, the construction translates coverability problems of TSA systems to finite sets of coverability problems of P/T nets.

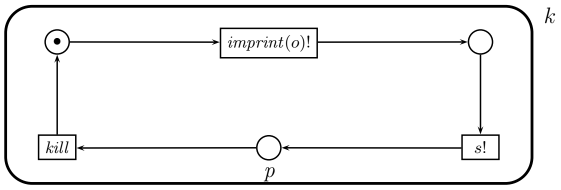

Fig. 8. TSA system

to analyze, and can be stated in terms of coverability (adding a control point to the specification), is checking that it is not possible for a controller to (for example) send out an alarm message without first been imprinted. If we assume that we have several nurses like the one in Fig. 2 and several controllers like the one in Fig. 3, we could also specify in terms of coverability the property that says that any nurse has at most one controller.

So far, we are regarding our TSA systems as the specification of systems using the resurrecting duckling policy, since our primitives assume that the imprinting and killing mechanisms are always correct. However, it is also desirable to perform a lower level analysis, with weaker assumptions. For that purpose, we will prove that every TSA system can be implemented by an MSPN system. An MSPN system intuitively consists on a set of Petri nets, each of them at some location. Those nets can move between location and, synchronize between them. Moreover, and this is more important, they can manage pure names (in the sense of [6]), that we call *identifiers*, and use them for authentication purposes, that is, as imprinting keys. Therefore, apart from ordinary tokens, MSPN components can have identifiers as tokens. For instance, a component could offer a synchronization only to components exhibiting a determined identifier. This is formalized by using variables as labels on arcs. MSPN components can also create fresh identifiers by means of a special variable ν, which can only be instantiated to fresh identifiers. However, they do not need to use this variable to implement TSA systems.

Proposition 3. *Every TSA system can be simulated by a MSPN system without name-creation transitions.*

Proof. Let $\mathcal{N}$ be a TSA system, that we can assume to be centralized. For every $N = (P, T, F, \lambda, o) \in \mathcal{N}$ we want to define a MSPN N^* in such a way that the MSPN system $\mathcal{N}^* = \{N^* \mid N \in \mathcal{N}\}$ implements $\mathcal{N}$. The main difference with the previous result is that components cannot have a global view of the dependencies in the system, but only of those that affect them directly, that is, they only know which components are their master and their slaves. For that purpose, we add a place *master* to every component, containing at each time the key used to communicate with its master. Similarly, if a net has capacity m then we add places *slave*$(1), \ldots,$ *slave*(m), used to communicate with the corresponding slaves. We represent alive components as those that contain a key in its *master* place. Since only alive components can execute actions, we must add that place as precondition of every transition (notice that components of

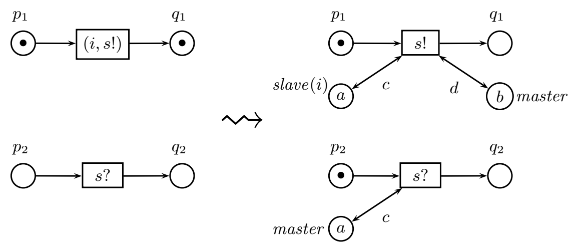

Fig. 9. Implementation of synchronizations

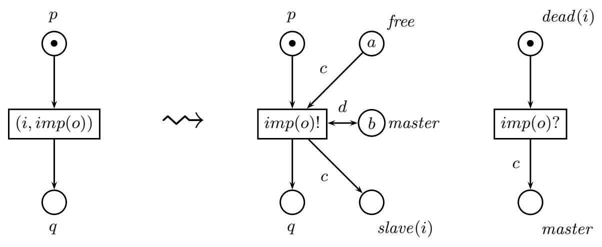

Fig. 10. Implementation of imprintings

type $\top$ are always alive though they do not have a master and hence they must have a key in that place, though they do not use it to communicate). We also add a place *dead*, marked with an ordinary token in mutual exclusion with *master*. Finally, we add a place *free* that contains a set of keys that are free to be used in future imprintings. If the component has capacity m and does not have any slave in the initial state, then that place must contain m pairwise different keys in that place in its initial marking.

Now let us see how we implement the transitions of each kind. Autonomous transitions are left as they are, except for the fact already mentioned that they have the *master* place as precondition and postcondition, so that they can only be fired when the component is alive and stay that way after it.

The synchronization of two transitions (see Fig. 9), labelled by $(i, s!)$ and $s?$, respectively, is implemented by the synchronization of two transitions, now labelled by $s!$ and $s?$. However, we want them to synchronize only when they share the proper dependency relation. For that purpose we add the place $slave(i)$ as precondition (and postcondition) of $(i, s!)$ and the place *master* as precondition (and postcondition of $s?$, and label all those arcs with the same variable. In this way, only components that contain the same value in those places (share the same key) can communicate.

Therefore, we have to guarantee that components only share values whenever told by the imprinting and killing primitives. In the implementation, imprinting and killing transitions are just ordinary synchronizing transitions, so that we

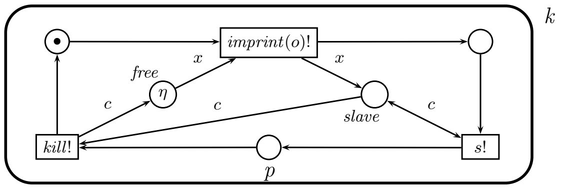

Fig. 11. Implementation of the TSA system in Fig. 8

assume there are synchronizing labels *kill* and *imprint*(o) for every $o \in \mathcal{O}$. The imprinting of a component of type o is implemented by the synchronized firing of two transitions, labelled by *imprint*(o)! and *imprint*(o)? (see Fig. 10). Notice that the latter transition can only be fired by dead components, so that in fact it is the only transition that has the place *dead* as precondition instead of *master*. When these two synchronize, a token is taken from the *free* place in the imprinting component and placed both in the corresponding *slave*(i) place and in the *master* place of the imprinted component.

The behaviour of the implementation of the killing transitions is dual, by means of the synchronization of two transitions labelled by *kill*! and *kill*?. Basically, the killing net moves the token in the corresponding *slave*(i) place and puts it in *free*, and the killed net removes its token from *master* and puts a black token in *dead*. However, the killing of a component causes the death not only of that component, but also of all its slaves. Therefore, before putting that token in place *dead* it must kill all its slaves.

MSPN systems without name-creation are still equivalent to P/T nets, so that we can take advantage of all the existing analysis methods for P/T nets even in the implementation of TSA systems as closed systems.

The previous proof profits from the fact that we are regarding TSA systems as closed systems, so that we can extensively and statically analyze all of their participants. In particular, we can check (and force) that the components being imprinted, that is, receiving their masters imprinting key, throw away those keys when they are killed. Therefore, components can safely reuse those keys in a forthcoming imprinting.

However, it is not very realistic to assume in the context of ubiquitous computing that we are dealing with closed systems. In this context, we want our services to be offered in a global and uniform way, so that some of the principals that imprint or some of the devices that are imprinted may not be part of the system, but part of the *environment* of the system. In this case, we can no longer check (even less force) that they erase the imprinting keys after been killed so that, if we reuse them, then the principals of the environment could use old keys to illegitimately interact with a component.

Let us again consider the simple TSA component in Fig. 8 or, more precisely, its implementation as an MSPN system (without name creation) in Fig. 11. This component attempts to imprint some device of type o, then uses some service s that this device provides and finally kills it, thus moving to its initial state. If this component is in an environment like the one in the left of Fig. 12 then its behaviour is the intended one. Indeed, when being killed by its master, that

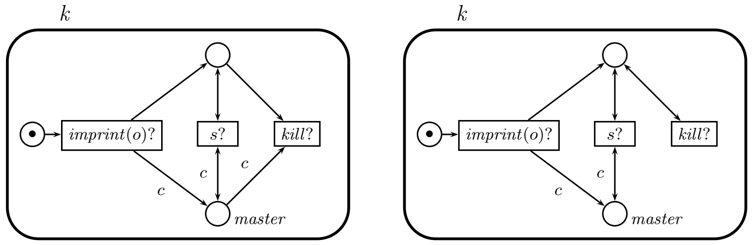

Fig. 12. Environments for TSA system in Fig. 8

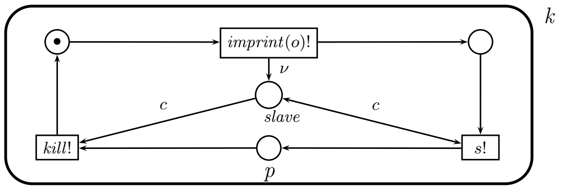

Fig. 13. Open implementation of the TSA system in Fig. 8

component throws away the imprinting key (the identifier in place *master*), so that it can be safely reused.

However, let us suppose that its environment is like the one in the right of the same figure, which is almost the same, except for the fact that it does not throw away that identifier (there is no arrow from place *master* to transition *kill*?) and can provide service s even after synchronizing in transition *kill* (which, in this case, does not have the effect of killing the component). Therefore, Fig. 11 would be a wrong implementation of the TSA system in Fig. 8, when considered in the environment in the right of Fig. 12. For more details about open MSPN systems see [16]. However, we can make use of the fresh-identifier creation capability of MSPN systems to perform a correct implementation, even in the open case.

Proposition 4. *Every TSA system can be simulated by an open MSPN system.*

Proof. The construction is identical to the one of the previous result, except for the fact that when impriting, keys are not taken from a place *free*, but created and when killing they are not put back in *free*, but erased.

Figure 13 shows the open implementation of the TSA system in Fig. 8, which is similar to that in Fig. 11, except for the fact that it does not have a place *free* for free identifiers and that it has the special variable ν labelling the arc to the place called *slave*. This special variable formalizes the creation of new identifiers, by forcing its instantiation only to identifiers that are not in the current marking. In particular, notice that the environment on the right of Fig. 12 can no longer offer service s after being killed, even if it does not throw away its key, since in a new imprinting the TSA component will use a different imprinting key in order to communicate with its slave.

As we have proved in [14, 16], the general class of MSPN system (in which we allow name creation) is not equivalent to P/T nets anymore, but are not Turing-complete either. In particular, we have proved that coverability is still decidable, even in the open case, so that we can still verify any property that can be stated in terms of it.

6 Conclusions and Future Work

In this paper we have defined a model based on Petri Nets that addresses the problem of Secure Transient Association in ubiquitous systems. The fact of using Petri Nets gives us an amenable and easy to grasp model.

Then we study whether we can approach the problem of the verification the properties of these systems. As a first step, we prove that TSA systems, seen as specification of systems using the primitives of the Resurrecting Duckling Policy, are equivalent to P/T nets, so that we have many decidable properties for them as reachability, coverability or home state.

Then we show how these primitives can be implemented solely relying on a mechanism that manages pure names, that can be seen as imprinting keys. These implementations can be studied from two different points of view: first assuming that our systems work in a closed environment, and then without this assumption. In the first case, we have proved that this implementation is still equivalent to P/T nets. However, this is no longer true in the second case, because we need to allow the creation of fresh identifiers, but even so we can approach the verification of their properties.

As future work, we plan to study in a more systematic way the kind of properties of TSA systems that can be studied. It would also be interesting to follow a dual approach, namely that in which we want to verify that a given system does indeed implement the resurrecting duckling policy, which would certainly lead us to approaches like proof carrying code [11] or dynamic typing [7]. We also plan to consider the studies about transactions in the context of Petri nets [1], to apply them to killing of slaves in our model, which are clearly transactions, though we do not treat them explicitly as such in the open implementation with identifiers.

Another direction for future work is that of extending the basic model with other features, such as allowing unbounded capacities for some componentes (e.g., users), other causes of death, or a finer treatment of localities.

In [15] we presented a tool for the verification of MSPN system, based on rewriting logic. We are currently extending this prototype to make it cope with the primitives presented in this paper, so that it automatically performs the translations we have described in the paper and check the properties that we have proved to be decidable using those translations.

References

[1] R. Bruni and U.Montanari. *Executing Transactions in Zero-Safe Nets.* 21st Int. Conf. on Application and Theory of Petri Nets, ICATPN'00. LNCS vol. 1825, pp. 83-102. Springer-Verlag, 2000.

[2] M. Carbone, M. Nielsen and V. Sassone. *A Calculus for Trust Management.* 24th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'04. LNCS vol. 3328, pp. 161-173. Springer-Verlag, 2004.

[3] J. Desel and W. Reisig, *Place/Transition Petri Nets.* Lectures on Petri Nets I: Basic Models, LNCS vol. 1491, pp. 122–173. Springer-Verlag, 1998.

[4] J. Esparza and M. Nielsen. *Decidability issues for Petri Nets - a survey.* Bulletin of the EATCS 52:244-262 (1994).

[5] D. Frutos Escrig, O. Marroquín Alonso and F. Rosa Velardo. *Ubiquitous Systems and Petri Nets.* Ubiquitous Web Systems and Intelligence, LNCS vol.3841. Springer-Verlag, 2005.

[6] A. Gordon. *Notes on Nominal Calculi for Security and Mobility.* Foundations of Security Analysis and Design,FOSAD'00. LNCS vol.2171, pp.262-330. Springer,2001.

[7] M. Hennessy and J. Riely. *Resource Access Control in Systems of Mobile Agents.* High-Level Concurrent Languages, HLCL'98. ENTCS 16:3–17. Elsevier, 1998.

[8] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.*. Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer, 1997.

[9] R. Milner, J. Parrow and D. Walker. *A calculus of mobile processes I.* Information and Computation 100(1):1–40. Academic Press Inc., 1992.

[10] R. Milner. *Theories for the Global Ubiquitous Computer.* Foundations of Software Science and Computation Structures-FoSSaCS 2004, LNCS vol.2987, pp.5-11. Springer-Verlag, 2004.

[11] G.C. Necula and P. Lee. *Safe, Untrusted Agents Using Proof-Carrying Code.* Mobile Agents and Security. LNCS vol. 1419, pp. 61-91. Springer, 1998.

[12] G.G. Richard. *Service Advertisement and Discovery: Enabling Universal Device Cooperation.* IEEE Internet Computing archive 4(5):18–26. IEEE Educational Activities Department, 2000.

[13] F. Rosa Velardo, O. Marroquín Alonso and D. Frutos Escrig. *Mobile Synchronizing Petri Nets: a choreographic approach for coordination in Ubiquitous Systems.* In 1st Int. Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, MTCoord'05. ENTCS, 150.

[14] F. Rosa Velardo, D. Frutos Escrig and O. Marroquín Alonso. *On the expressiveness of Mobile Synchronizing Petri Nets.* In 3rd International Workshop on Security Issues in Concurrency, SecCo'05. ENTCS (to appear).

[15] F. Rosa-Velardo. *Coding Mobile Synchronizing Petri Nets into Rewriting Logic.* 7th Int. Workshop on Rule-based Programming, RULE'06. ENTCS (to appear).

[16] F. Rosa-Velardo, and D. Frutos-Escrig. *Symbolic Semantics for the Verication of Security Properties of Mobile Petri Nets.* 4th Int. Symposium on Automated Technology for Verification and Analysis, ATVA'06. LNCS vol. 4218, pp. 461-476. Springer, 2006.

[17] F. Stajano and R.J. Anderson. *The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks.* 7th Int. Workshop on Security Protocols. LNCS vol. 1796, pp. 172-194. Springer, 1999.

[18] F. Stajano. Security for Ubiquitous Computing. Wiley Series in Communications Networking & Distributed Systems. John Wiley & Sons, 2002.

[19] R. Want. *Enabling Ubiquitous Sensing with RFID.* Computer vol.37(4), pp.84-86. IEEE Computer Society Press, 2004.

[20] M. Weiser. *Some Computer Science Issues in Ubiquitous Computing.* Comm. of the ACM vol.36(7), pp.74-84. ACM Press, 1993.

[21] M. Weiser. *The Computer for the 21st Century.* In Human-computer Interaction: Toward the Year 2000, pp.933-940. Morgan Kaufmann Publishers Inc, 1995.