Focusing on Mobility

Klaus Bergner, Radu Grosu, Andreas Rausch,
Alexander Schmidt, Peter Scholz, Manfred Broy
Institut für Informatik, Technische Universität München, Germany
http://www4.informatik.tu-muenchen.de

Abstract

In this paper, we motivate the importance of the field of mobile computing and survey current practical and formal approaches. We argue that the existing formalisms are not sufficiently general and powerful because they do not model all necessary concepts of mobility adequately. The main contribution of the paper is, therefore, to identify and define the fundamental concepts of mobile systems by providing a precise, mathematical foundation. The model we present is an extended variant of existing, compositional network models for control and data flow of non-mobile systems, enriched by the concept of locations as places containing components. To model the migration of a component from one location to another, the containment relation may change dynamically over time. Based on this formal model, we define a number of fundamental properties and characteristics such as network transparency. Finally, we demonstrate how existing description techniques may be extended in the context of mobility, and sketch a supporting CASE tool.

1 Why Mobility is Important

Recent developments in hardware and software technology have created the infrastructure for a new computation paradigm: mobile computing. According to this paradigm, both hardware and software components may dynamically migrate between distributed locations.

Mobility implies a fundamental paradigm shift in the area of computing. According to Carlo Ghezzi [1], for example, “the long term vision is that computers are no more viewed as mainly autonomous and self-contained computing devices accessing local resources, occasionally communicating with each other; rather, they are part of a global computing platform, built upon a synergy of local and remote resources, whose sharing is enabled by broadband communication networks”. Without doubt, mobile computing will enable a wealth of novel applications, for example, in the area of fieldwork construction, maintenance, and life-cycle management systems, as well as smartcard-based electronic commerce applications.

In the following, we sketch some of the possibilities in aviation. During the production of an airplane, the respective manufacturers may integrate small computers with sensors into the single parts of the aircraft to trace the location of the parts during construction (which is performed by workers with wearable computers, of course).

Once the airplane is completed, the integrated sensors provide actual information over the state of the parts, enabling proactive maintenance and replacement on need. During flight, the communication between its computers and stationary computers on the ground (and also with the mobile computers of the maintenance staff) has to be organized. Finally, the passengers and their various mobile computing devices must eventually be taken into account, ranging from smartcards used to pay calls over the airplane’s phone system to laptop computers connected to the internet via the airplane’s local area network.

Scenarios like this involve a variety of different hardware and software components with various, dynamically changing relations between them.

Currently, an integrated view on mobility is still missing. Research and development is mostly focused on relatively isolated areas, like mobile hardware design, mobile data management [2, 3], and protocols for mobile and wireless networks [4]. These areas are exploited by integrated technical platforms for the development of mobile systems. Such platforms try to reach a high degree of mobile transparency, so that programmers are able to implement a mobile system as they are used to implement a non-mobile system. There are three groups of platforms:

- The platforms in the first group are based on traditional programming platforms for distributed systems, like CORBA [5] and DCOM [6]. Based on their facilities for transparent communication, support for mobile code and mobile data may be implemented [7].

- The second group of platforms has its roots in the area of distributed operating systems. Odyssey [8], for example, enhances the distributed operating system Mach [9] with additional features like hierarchical namespaces and code as well as data resources. Mobile applications may then navigate within the namespaces and get access to various resources.

- The last group of mobile system platforms are mobile languages. Most languages are based on rather traditional concepts, enhanced by a few constructs for process migration from one host to another. Well-known implementations of mobile code languages are Telescript [10], Java [11], Aglets [12], Javelin [13], Agent Tcl [14], Sumatra [15], Omniware [16], Obliq [17], ML [18], TACOMA [19], and Ara [20].

Most current platforms do not cover all aspects of mobility, but concentrate on single aspects, like the mobility of data and code, the concurrency of mobile system components, or the relocation of mobile devices. A truly integrated approach supported by an overall methodology for the development of mobile systems is still missing. In our opinion, this is partly due to the absence of precisely defined concepts and formalisms serving as a foundation for advanced languages, development techniques, and tools. An adequate formalism for mobility would also contribute to the reliability of mobile systems, as it is a necessary precondition of formal correctness proofs.
In Section 2, we survey the existing formalisms in the area of mobile computing and identify some of their deficiencies. Section 3 presents our formal approach, which is based on existing compositional models for control and data flow of dynamic, non-mobile systems. We show that the basic formalism can be extended and adapted to special applications. Finally, Section 4 demonstrates how existing description techniques and CASE tools for non-mobile systems may be extended in order to deal with mobility. A short conclusion ends the paper.

2 Existing Formal Approaches

As explained in the previous section, a clear, well-understood formal foundation of mobile systems is currently missing. This is mainly due to the complexity of these systems and the multitude of involved concepts:

- First, mobile systems are *interactive* because they have to be able to continuously exchange messages with their environment. As a consequence, they imply a shift from sequential systems to concurrent systems.

- Second, mobile systems may evolve *dynamically* over time. Even basic system properties like the number of components and the structure of the connections between them may change at runtime.

- Third, mobile systems involve components that *migrate* from one location to another location.

Based on these characteristics, we identify three layers of increasing complexity, namely interactive systems, dynamic systems, and mobile systems. In the following, we present some typical examples of formal approaches in these three layers. The chosen set of formalisms is by no means complete—it merely serves as an illustration of the main features (and the complexity) of mobile systems.

2.1 Interactive Systems

CCS and CSP [21, 22]: These two approaches formalize *static*, *interactive systems* in a rather operational fashion. While CCS is purely operational and equivalence between processes is defined only by bisimulation, CSP also has a denotational failure semantics. Both approaches use a synchronous *handshaking* communication—the sender is blocked as long as the receiver is not ready to communicate.

FOCUS [23, 24, 25, 26]: With respect to mobility, FOCUS is at the same level as CCS and CSP because it only characterizes *static, interactive systems*. However, in contrast to the other two approaches, FOCUS is mainly a denotational approach. It is based on well-established techniques for describing the behavior of sequential systems as a *relation* between input and output values [27, 28]. In contrast to these formalisms, it provides the concept of *time* by using relations between sequences of input and output values. Communication is asynchronous—a sender is never blocked by a receiver.

2.2 Dynamic Systems

π-Calculus and HO-π-Calculus [29, 30, 31]: The π-calculus is an extension of CCS to model *dynamic reconfiguration*. This feature is achieved by allowing channel identifiers to "move" along channels. This is quite similar to familiar object-oriented languages where object identifiers may be passed between programs. In higher-order (HO) π-calculus, not only channel identifiers, but also whole processes may "move" along channels. This corresponds to passing clones in object-oriented languages. As in CCS, communication is done synchronously using handshaking.

DYNAMIC FOCUS [32, 33, 34, 35]: This approach extends FOCUS similar to the way π-calculus extends CCS by allowing channel names to be passed between processes. This leads to a new, denotational understanding of *dynamics* as a set of *privacy invariants* that have to be maintained by the interacting processes. As in FOCUS, communication in DYNAMIC FOCUS is asynchronous.

Blue Calculus [36]: The blue calculus is an *asynchronous* version of Milner's π-calculus, based on the idea that the messages are elementary processes that can be sent without any sequencing constraint. Furthermore, names may be restricted, making them private in the context of an agent.

These two concepts are sufficient to encode the synchronous communication of the π-calculus.

Chemical Abstract Machine [37]: The chemical abstract machine is a semantic framework based on the chemical metaphor used in the Gamma language of Banatre and Le Métayer. States of a machine are understood as chemical solutions where molecules may interact according to reaction rules. It is possible to dynamically encapsulate subsolutions within membranes that force reactions to occur locally. Using the abstract machine, the operational semantics of the π-calculus machine used by Milner to encode the lambda-calculus is defined.

2.3 Mobile Systems

Join-Calculus [38]: The join-calculus is an experimental language based on the homonymous process calculus, providing basic support for distributed programming. The join-calculus programming model features concurrent processes running on several machines, static type-checking, global lexical scope, transparent remote communication, agent-based mobility, and some failure-detection.

Ambient-Calculus [39]: An ambient is a bounded place of computation. Each ambient has a name, a collection of subambients, and a collection of local agents. These are computations that run directly within the ambient. The model of mobile ambients is an extension of the asynchronous π-calculus. It supports reasoning about mobility and also about system security. Note however, that the ambient-calculus does not fundamentally distinguish between components and locations, as it defines only a single subambient relation.

The above survey presents a rather balanced number of approaches in each conceptual category. An extensive survey would reveal a pyramid—while there are many formalisms for interactive systems, only few approaches for dynamic and especially mobile systems yet exist.
It is interesting to note that distribution, which is necessary to model mobility, brought new requirements concerning communication primitives. Whereas synchronous communication by handshaking is quite natural for local systems, it turned out to be a great restriction for distributed systems. This is the reason why both the join-calculus and the ambient-calculus are *asynchronous*. Moreover, the introduction of a *location* concept seems to be essential to model process migration.
Most of the published approaches are operational, making the translation to programming languages easy. However, there is no doubt that a denotational understanding of mobility is indispensable for a simple characterization of process equivalence and compositionality.

3 A Model for Mobile Systems

The proposed model is a mobile extension of the existing, compositional FOCUS network model with a denotational semantics. We first outline its essential concepts in an informal way. In the following, the concepts and relations are explained in a more formal way by giving mathematical definitions. Finally we show some variations of our formalism and consider extension concepts.

3.1 Essential Concepts of Mobile Systems

We think that an explicit model for mobile systems must distinguish mobile entities, namely, *components*, from the *locations* between which they are moving. Surprisingly, this seemingly simple and evident distinction is only implicitly present in some of the current formalisms for mobility, as can be seen from Section 2. Components may be software as well as hardware components, ranging from software agents and Java programs over laptops, cellular phones, and airplane onboard computers to persons moving around in an 'intelligent' building, for example. Locations may be physical locations in the real world, like rooms of a building, streets in a town, or squares of a wooden chessboard, but they may also be logical or conceptual locations, like nodes of a logical network, departments of a company, or squares of a simulated chessboard.
The most important relation between components and locations is *containment*. A location *contains* a number of components at a certain time. In a mobile system, the containment relation changes over time. A cellular phone contained in a certain cell at a certain time, for example, may later on leave this cell and migrate to another cell, thereby changing the containment relation.
Furthermore, we allow hierarchical components by introducing a *subcomponent* relation between components. In the *subcomponent* relation, each subcomponent may only have at most one parent component in order to ensure proper encapsulation of subcomponents.
Locations may also be hierarchical. However, in contrast to components, there may be multiple *sublocation* relations, and a location may be a sublocation of multiple parent locations. However, each *sublocation* relation must be a directed acyclic graph—a location may not be its own sublocation. Examples for sublocation relations are the spatial and logical inclusion of a kitchen or bathroom in a flat, the logical inclusion of a subnet in a computer network, and the mathematical inclusion of a two-dimensional vector space in a three-dimensional vector space.
Finally, we also introduce a *room* relation which assigns locations to components. Examples for this relation are the relation between the location inside a sugar bowl and the sugar bowl itself, and the relation between a computer's mem-

ory (understood as a location containing program components) and the computer itself. Note that the relations $room$ and $contains$ are not inverse—the memory of the computer doesn't contain the computer itself. Note, furthermore, that our model yields two possibilities for component inclusion: besides being a subcomponent of another component (like the keyboard of the computer), a component may be contained in a location assigned to a component (like the programs living inside the memory assigned to the computer component).
Of course, the essential concepts identified above are not sufficient to model all properties and features of mobile systems. However, we think that additional concepts should be introduced based on this foundation in order to tailor it to various applications (cf. Section 3.3). An example is the concept of a *way*, defined as a connection between locations. We have not introduced ways as a base concept because it is not necessary for systems with freely migrating components. Furthermore, there are many ways to introduce ways—they may constrain or influence, for example, the migration of components or the communication between them.

3.2 Formalizing the Concepts

The mathematical model presented in the following subsections is structured according to the three layers explained at the beginning of Section 2: starting with a model for static, interactive systems, we obtain models for dynamic and, finally, mobile systems by adding dynamicity and locations, respectively.

3.2.1 Static Systems

We model an *interactive system* by a network of *autonomous components* which communicate via *directed channels* in a *message asynchronous* way. Message asynchrony means that it is always possible to send a message along a channel. Figure 1 shows a simple component (visualized by a box with a solid borderline) with input channels i_1 to i_m and output channels o_1 to o_n.

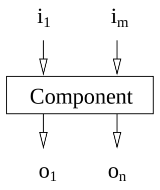

Figure 1: Component with input and output channels

Syntactic Interface: Let $ChanId$ be the set of *channel identifiers*, let D be the set of *messages* flowing along channels[1] and let $I, O \subseteq ChanId$ be the identifiers of the *input* respectively *output channels* of components. The *syntactic interface* of a component is then given by the pair of *labelled product types* $I \to D$ and $O \to D$. For simplicity, we denote these product types also by $\vec{I}$ and $\vec{O}$, respectively.

In order to define and analyze the flow of messages along the channels, we have to model *time*. The simplest assumption about time is that it increases with some constant time unit, i.e., that it is *discrete*. For convenience, we use $\mathbb{N}$ as abstract time axis. In this case, the messages exchanged over time along a channel define an *infinite sequence* $s \in D^{\mathbb{N}}$, where $D^{\mathbb{N}}$ is an abbreviation for $\mathbb{N} \to D$, such that $s(i)$ is the message occurring in time unit i. We call such a sequence a *channel history*.
An assumption leading to a simple and uniform model is that the message flow is *time synchronous*, i.e., that time flows in the same time scale for each channel. In this case, the labelled product type $I \cup O \to D^{\mathbb{N}}$ of the histories of a component's channels is equal to the history type of the component's interface $(I \cup O \to D)^{\mathbb{N}}$. As a consequence, $I \to D^{\mathbb{N}}$ is equal to $(I \to D)^{\mathbb{N}}$, abbreviated to $(\vec{I})^{\mathbb{N}}$. The same holds for the output channels $(\vec{O})^{\mathbb{N}}$. Note that this is not true for other, non-synchronous models of message flows. Further notice that the limitation to a time synchronous model is no restriction at all in practice, as time synchrony is a very powerful model. Based on it, other models of time can be easily defined, including, for instance, asynchronous ones.

Semantic Interface: Due to the equivalence of $I \cup O \to D^{\mathbb{N}}$ and $(I \cup O \to D)^{\mathbb{N}}$ for synchronous time, the behavior of a component can be completely described by a sequence of *input/output relations*

$$r \subseteq (\vec{I} \times \vec{O})^{\mathbb{N}}.$$

We call this relation sequence the *semantic interface* of the component. We assume that messages occurring in the histories of the output channels in time unit i are independent of the future input history, i.e., from the messages occurring in the histories of the input channels at time $j > i$. Clearly, each implementable component must obey this restriction. We call behaviors respecting this property *time guarded*. We define

$$Cmp = (\vec{I} \times \vec{O})^{\mathbb{N}}$$

to be the set of time guarded sequences of input/output relations. There are multiple techniques to define these relations. A very common one are *state transition diagrams* (STD) describing a kind of automata consuming input and producing output with each transition.

[1]Assuming different sets of messages for different channels is surely necessary for any realistic system. However, it complicates the formal treatment and does not provide further insights to the considered model.

3.2.2 Hierarchical Components

Components may be assembled to more complex components hierarchically. If $CompId$ is the set of *component identifiers*, hierarchy may be characterized by defining a *subcomponent relation*

$$subcomp \subseteq CompId \times CompId$$

associating with each component identifier the set of its (sub)component identifiers.
The identifier c of a leaf component is mapped to a triple (I, O, r) of its input/output channels and its input/output relation sequence by using the relations

$$in, out \subseteq CompId \times ChanId$$

and the function

$$beh : CompId \to Cmp$$

defined as follows:

$$in(c) = I, \quad out(c) = O, \quad beh(c) = r$$

As Figure 2 shows, a set of components may be combined, yielding a hierarchical component. Its interface consists of the channels of its inner components that are not fed back to other inner components.

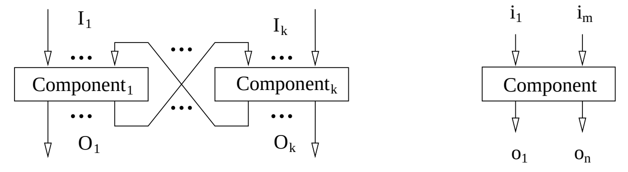

Figure 2: Assembling a hierarchical component

Let c be the identifier of a hierarchical component with $subcomp(c) = \{c_1, \ldots, c_n\}$. Then we define in, out and beh for c as follows:

$$\begin{array}{rcl} in(c) & = & (in(c_1) \cup \ldots \cup in(c_n)) \setminus \\ & & (out(c_1) \cup \ldots \cup out(c_n)) \\ out(c) & = & (out(c_1) \cup \ldots \cup out(c_n)) \setminus \\ & & (in(c_1) \cup \ldots \cup in(c_n)) \\ beh(c) & = & \{(i, o) \mid \exists q.(i \uplus q, o \uplus q) \in beh(c_1) \uplus \\ & & \ldots \uplus beh(c_n)\} \end{array}$$

The sets of input channels I and output channels O of the hierarchical component c consist of the union of the subcomponent sets of input channels I_k and output channels O_k, respectively, minus the set of channels $P = (\cup_{k=1}^{n} in(c_k)) \cap (\cup_{k=1}^{n} out(c_k))$ that are fed back (see Figure 2, left). The feedback channels are private and therefore hidden.
The behavior of a hierarchical component c is the set of labeled input/output history pairs $(i, o) \in (\vec{I})^{\mathbb{N}} \times (\vec{O})^{\mathbb{N}}$ obtained from the associative product $\uplus$ of the subcomponent behaviors[2] by hiding the histories $q \in \vec{P}^{\mathbb{N}}$ which are fed back. By $i \uplus q$ and $o \uplus q$ we denote the unique tuples constructed from i, o and q such that $i \uplus q \in (\vec{I})^{\mathbb{N}} \uplus (\vec{P})^{\mathbb{N}}$ and $o \uplus q \in (\vec{O})^{\mathbb{N}} \uplus (\vec{P})^{\mathbb{N}}$. At this time, we must ensure that feedback loops contain a delay. This can be achieved by components that need at least one time tick to produce their output.
Note that both for leaf components as well as for composed components the syntactic interface given by in and out does not change over the whole communication history. As a consequence, their interconnection structure is also fixed. We therefore call systems characterized by such components *static.*

[2]Note that the associative product $\uplus$ of two labeled products is a labeled product having as set of labels the union of the component sets of labels.

3.2.3 Dynamic Systems

There are two orthogonal ways to change a system *dynamically*. First, the *number of components* may change. For example, the interconnection structure of the interactive queue shown in Figure 3 is fixed, even if the number of cells may vary over time. In such systems, the *syntactic interface* of the involved components remains fixed. However, these components are *recursive.*

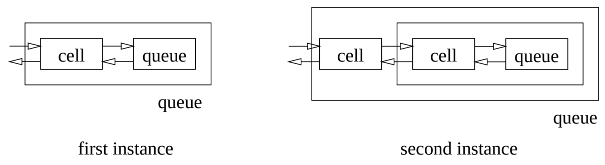

Figure 3: An interactive queue

Second, the *interconnection structure* may change, as shown in Figure 4. Here, the components remain the same, but the connection structure changes. In such systems, the syntactic interface of the involved components changes over time. Changes of the connection structure are usually achieved by passing channel identifiers between components. At each moment, the set of channel identifiers owned by a component determines its syntactic interface and, consequently, its communication capabilities.

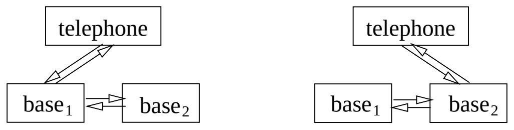

Figure 4: A mobile telephone

The syntactic interface of a dynamic component may vary in time, yielding the dynamically changing sets I_t and O_t, respectively. For each time unit t, the relation $r_t \subseteq (\vec{I_t} \times \vec{O_t})$ gives the behavior of r in that time unit. As a consequence, the overall input/output relation r that reflects the complete input/output history of a dynamic component has the following type:

$$r \subseteq ((ChanId \to D) \times (ChanId \to D))^{\mathbb{N}}$$

where $ChanId \to D$ denotes the set of labeled tuples with labels in a subset of $ChanId$. The functions in and out are also not constant anymore. For each moment of time t, $in_t(c) = I_t$ and $out_t(c) = O_t$ are the identifiers of known (or accessible) input and output channels, respectively. Similarly, the subcomponent relation $subcomp$ may also vary in time. For each time unit t, $subcomp_t(c) = \{c_1, \ldots, c_{n_t}\}$ gives the set of current subcomponents of c, where n_t is a natural number that depends on t. Hence,

$$\begin{array}{lll} in & \subseteq & (CompId \times ChanId)^{\mathbb{N}} \\ out & \subseteq & (CompId \times ChanId)^{\mathbb{N}} \\ subcomp & \subseteq & (CompId \times CompId)^{\mathbb{N}} \end{array}$$

The relations in, out, and beh are defined for composed components by extending their static counterparts over time as shown below:

$$\begin{array}{lll} in_t(c) & = & (in_t(c_1) \cup \ldots \cup in_t(c_{n_t})) \backslash \\ & & (out_t(c_1) \cup \ldots \cup out_t(c_{n_t})) \\ out_t(c) & = & (out_t(c_1) \cup \ldots \cup out_t(c_{n_t})) \backslash \\ & & (in_t(c_1) \cup \ldots \cup in_t(c_{n_t})) \\ beh_t(c) & = & \{(i, o) \mid \exists q.(i \uplus q, o \uplus q) \in beh_t(c_1) \uplus \\ & & \ldots \uplus beh_t(c_{n_t})\} \end{array}$$

3.2.4 Hierarchical Locations and Containment

In order to describe mobile systems, we introduce *locations*. Intuitively, a location is defined as a place where components may be located and where computation takes place.
As with components, we also allow hierarchical locations. However, in contrast to components, a location may be a sublocation of many other locations, yielding a DAG (directed acyclic graph) structure. We define the sublocation relation as follows:

$$subloc \subseteq LocId \times LocId$$

There may also be more than a single sublocation relation in order to model different location hierarchies, for example, a spatial and a logical one. To keep the discussion simple, the following definitions consider only a single relation. The extension to multiple hierarchies is straightforward.
Each component is assigned to the set of locations where the component resides via the $contains$ relation. Conversely, locations can be assigned to components via the $room$ relation:

$$\begin{array}{lll} contains & \subseteq & LocId \times CompId \\ room & \subseteq & CompId \times LocId \end{array}$$

Figure 5 shows examples for these relations. The notation of the diagram on the left side uses dotted lines to distinguish locations from components, which are drawn with solid lines. Note that the $room$ relation between the location ε and the component a is not visualized on the right side to keep the diagram simple.

3.2.5 Mobile Systems

In a mobile system, each of the relations introduced in the previous section may change over time. We therefore get the following time-varying relations:

$$\begin{array}{lll} subloc & \subseteq & (LocId \times LocId)^{\mathbb{N}} \\ contains & \subseteq & (LocId \times CompId)^{\mathbb{N}} \\ room & \subseteq & (CompId \times LocId)^{\mathbb{N}} \end{array}$$

For each moment of time t, location l, and component c, $subloc_t(l)$ yields the sublocations of l, $contains_t(l)$ yields the set of components contained by the location l, and $room_t(c)$ yields the set of locations contained by the component c.
If these relations do not vary over time, we obtain the classical conceptual framework of distributed systems. To call a system *mobile*, it is sufficient if at least one of these relations varies over time. Usually, this is the case for the $contains$ relation.

3.2.6 Network Transparency

Component migration is usually achieved by passing the component's closure, i.e., the component's behavior and state, between locations. In our model, migrating components also keep their communication capabilities, represented by their channels. As can be seen from Figure 6, migrating the *laptop* component therefore does not involve any changes of syntactic interfaces.
If no further constraints or assumptions apply, the subcomponent hierarchy and the timed, functional behavior of a mobile system are not influenced by changes of the $subloc$, $contains$, and $room$ relations at all. We call this property *strong network transparency*. In most cases, this property is too strict because it constrains not only the functional behavior, but also the timing of the system. A more reasonable property is called *weak network transparency*. It only requires that the functional behavior according to some specification remains stable and allows the timing to vary according to the actual configuration of the system components. If also the functional behavior may vary, we call the system *network aware*.

3.3 Properties and Possible Variations

Based on the formal foundation presented in the previous section, many specialized variations and properties may be defined:

Flat versus Leveled Structures: If no sublocation or subcomponent relations are defined, the result is a flat location

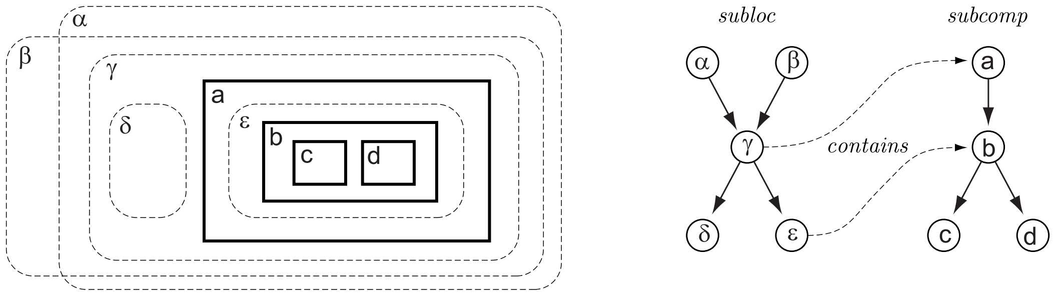

Figure 5: Sublocation and subcomponent hierarchies

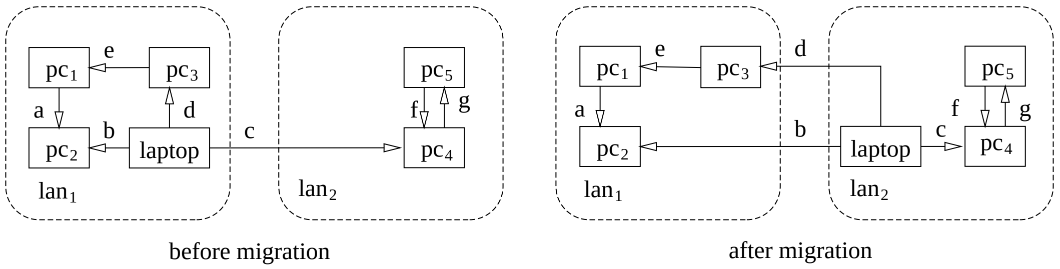

Figure 6: Migration of a laptop

or component structure. This allows to model simple scenarios, like the migration of unstructured software agents in a flat structure of computing nodes, without introducing unnecessary complexity. In a system model with flat structures $subloc = \emptyset$ and $subcomp = \emptyset$ must hold, respectively.

Tree Structures versus General DAG Structures: If additional complexity is required, arbitrarily complex sublocation relations may be defined. In many cases, tree-like relations will be sufficient, for example, to model the relation between a building and its rooms. However, a room may also be situated in many other rooms. This applies, for example, to a street that belongs to different quarters of a city. Although the street could be modeled with a tree structure by dividing it up into its single segments and treating them as sublocations of the different city quarters, it is often more natural to view the street as a whole. If a system model has a tree structure, each sublocation has a single parent location, correspondingly $\forall loc \in LocId : |subloc^{-1}(loc)| \leq 1$ must hold.

Single Hierarchy versus Multiple Hierarchies: In many cases, a single location structure is sufficient to model all interesting aspects of reality. However, there are also situations in which multiple location structures are necessary. Imagine, for example, a cellular phone which is able to change its spatial position as well as its logical network cell. In this case, the phone component is contained in two locations of different location hierarchies at the same time. To support multiple hierarchies in the system model we have to introduce a family of location relations $subloc_i \subseteq (LocId \times LocId)^{\mathbb{N}}$. In the case of the cellular phone, the system model would include two location relations: $subloc_{logical}$ and $subloc_{spatial}$.

Single versus Multiple Containment: There are simple systems whose components are contained in exactly one location of a given location hierarchy at each time. This applies, for example, to non-distributed software agents which are executed on single computing nodes. To model single containment, the $contains$ relation must obey the following condition: $\forall comp \in CompId : |contains^{-1}(comp)| \leq 1$. However, there are scenarios where single containment is not sufficient. Imagine, for example, a huge anaconda component creeping out of the living room location of a little flat, its head entering the bedroom location, while its tail still has not left the bathroom location. In this case, it would be difficult to use a single-containment model because the snake can not be easily cut into segments that could be assigned clearly to a single location.

Static versus Dynamic Structures: The $contains$ relation is usually changing over time, while the $room$ relation will often be static (in case of the sugar bowl component, for example, its inside location can hardly be removed or changed without destroying the sugar bowl component). For the sublocation and/or subcomponent relations, both static and dynamic relations make sense. To define a sublocation relation modeling the spatial structure of the three-dimensional space, for example, a static structure seems to be adequate, unless we consider relativistic effects like the distortion caused by gravity. In contrast to this, containment structures will usually be dynamic. An example is, again, the anaconda component, which, after arriving in the bedroom location, may incorporate the little woman component contained there, thus changing the subcomponent relation.

Free versus Constrained Migration and Communication As mentioned in Section 3.1, migration may be constrained by *ways* in some systems. Components may, for example, migrate only if there exists a suitable way leading from its present location to the new location. Ways could also influence the speed of migration or even the behavior of the components migrating over them. Similar considerations may apply to communication: channels between components at different locations may only exist if there is a suitable way, and the communication delay may be influenced by the length of the way, for example.

Additional Properties To extend the formal model, arbitrary kinds of additional relations may be added. Examples are relations like, for example, *dependsOn*, *cooperatesWith*, *executedOn*, *availableAt*, or *slowsDownItsNeighboursByFactorTen*. Like the pre-defined relations, these relations may hold temporarily or permanently, and they may vary over time. Often, additional relations will violate the network transparency and latency tolerance of a system, so that additional measures have to be taken to ensure its correct operation.

4 Tool Support

In general, there are two different groups of tools for mobile systems: *management tools* and *CASE tools*.
Management tools are used to perform installation, configuration and maintenance of already implemented mobile systems. The *Productivity Tools for Managing Distributed Applications* from Inprise [40], for example, can be used to browse and change the current locations of CORBA objects. Another example is a location management tool which could be used by a cellular phone company to restructure the location hierarchy during run-time of the system, for instance, when new transmitting stations are built or when a license for a new region is acquired. Management tools could also log the migration of components to figure out bottlenecks.
To our knowledge, CASE tools tailored to the specific needs of mobile systems are still missing. It seems to be promising to enhance and adapt existing CASE tools for embedded systems. The static version of the development method FOCUS (cf. Section 2) is supported by a prototype implementation of a multi-user software engineering tool for the specification and simulation of distributed systems, named AUTOFOCUS [41]. The concepts and description techniques of AUTOFOCUS are described in detail in [42]. Currently, AUTOFOCUS does not support description techniques for mobile systems.
To fully support mobile system development in AUTOFOCUS, the existing description techniques have to be extended with new concepts. Extended event traces, a subset of Message Sequence Charts as standardized in ITU Z.120 [43], could be enriched as shown in Figure 7. It shows the same components and locations as Figure 6. In the beginning, the pc1, pc2, pc3, and laptop components are contained in the lan1 location, while the pc4 and pc5 components are containd in the lan2 location. The rest of the diagram is similar to UML sequence diagrams [44], with the additional concept of migration, which is represented by a bold arrow. In the scenario, the laptop component first reads all data from the PCs in lan1 into its local memory. Then it migrates to lan2 and stores the data.
Apart from the description techniques, the simulation engine of AUTOFOCUS also needs to be adapted for mobile systems [45]. Concepts like locations, migration, and communication latency have to be integrated into the simulator.

5 Conclusion

In this paper, we have surveyed the state of the art in the field of mobile computing, especially with respect to its formalization. In our opinion, the existing formalisms are not sufficient, partly because they do not distinguish adequately between components and locations. Our own approach is based on the established and well-developed formal model for static and dynamic systems of the FOCUS method.
Our main contribution is the introduction of *locations* as the places where components are situated and between which they migrate. We have also outlined how to extend and enrich the basic formalism by additional concepts. Of course, the foundation presented in the paper can only serve as a starting point—specialized models, calculi, and languages must be elaborated that reflect the characteristics of real-world applications. This pertains especially to graphical description techniques which are particularly needed for communication with end users. To evaluate the usefulness of these languages and techniques, we will perform a reasonably complex case study.
Our long-term goal is to develop a general development method for mobile systems based on the formal foundation presented in this paper. Besides graphical description techniques and a suitable development process model, such an overall method should also contain a stock of well-proven standard solutions and architectures which can be used for the construction of the complex, mobile systems of the future.

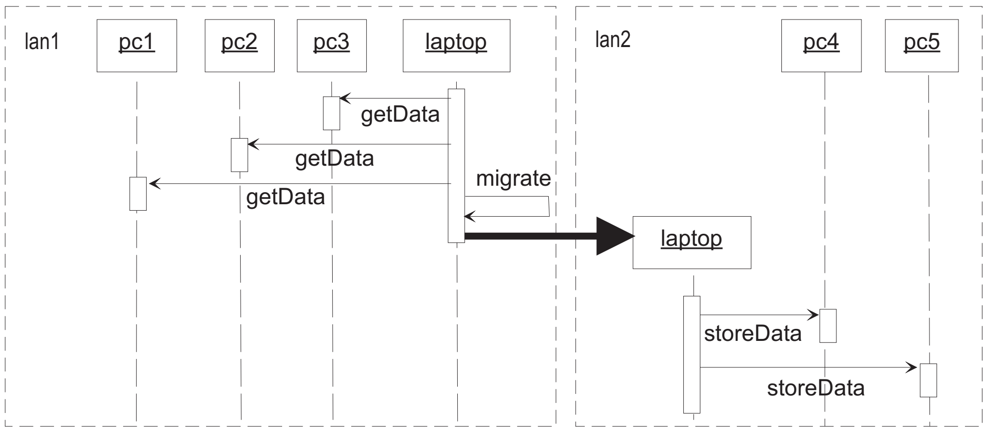

Figure 7: Sequence diagram with mobile component

Acknowledgements

We thank Alexander Vilbig for helpful comments on earlier versions of this paper.

References

[1] G. Cugola, C. Ghezzi, G. P. Picco, and G. Vigna, "Analyzing mobile code languages," in *Mobile Object Systems: Towards the Programmable Internet*, no. 1222 in LNCS, Springer-Verlag, 1997.

[2] Transarc Corporation, "Home page, `http://www.transarc.com`," 1998.

[3] M. Satyanarayanan, M. R. Ebling, and J. Raiff, *Coda File System: User & System Administrator's Manual*, 1995.

[4] J. Ioannidis and G. Q. Maguire, "The design and implementation of a mobile internetworking architecture," in *Proc. of the Winter 1993 USENIX Conference*, (San Diego, California), pp. 489–502, 1993.

[5] Object Management Group, "OMG website, `http://www.omg.org`."

[6] K. Brockschmidt, *Inside OLE2*. Microsoft Press, 2nd ed., 1995.

[7] Objectspace, Inc., "Voyager home page, `http://www.objectspace.com`," 1998.

[8] M. Satyanarayanan, B. Noble, P. Kumar, and M. Price, "Application-aware adaptation for mobile computing," 1994.

[9] R. Rashid, "From rig to accent to mach: The evolution of a network operating system," in *Fall Joint Computer Conference*, pp. 1128–1137, 1986.

[10] J. White, "Mobile agents white paper `http://www-iiuf.unifr.ch/˜chantem/white_white paper/whitepaper.html`," 1996.

[11] D. Flanagan, *Java in a Nutshell*. O'Reilly & Associates, Inc., 2nd ed., 1996.

[12] B. Venners, "Under the hood: The architecture of aglets, `http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html`," *JavaWorld*, April 1997.

[13] P. Cappello, B. Christiansen, M. Ionescu, M. Neary, and K. Schauser, "Javelin: Internet-Based Parallel Computing Using Java," in *ACM Workshop on Java for Science and Engineering Computation, Las Vegas*, ECOOP, June 1997.

[14] R. S. Gray, *Agent Tcl: Alpha Release 1.1*, 1995.

[15] A. Acharya, M. Ranganathan, and J. Saltz, "Sumatra: A language for resource-aware mobile programs," in *Mobile Object Systems: Towards the Programmable Internet*, pp. 111–130, Lecture Notes in Computer Science No. 1222, April 1997.

[16] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and language-independent mobile programs," in *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pp. 127–136, ACM, May 1996.

[17] L. Cardelli, "A language with distributed scope," in *Computing Systems*, pp. 27–59, 1995.

[18] D. Duggan and P. Przybylski, "A type-based implementation of language with distributed scope," in *Proceedings of the 2nd ECOOP Workshop on Mobile Object Systems*, pp. 52–59, ECOOP, July 1996.

[19] D. Johansen, R. van Renesse, and F. B. Schneider, "An introduction to the TACOMA distributed system," Tech. Rep. 95-23, Department of Computer Science, University of Tromso, 1995.

[20] H. Peine, "An introduction to mobile agent programming in the Ara system," Tech. Rep. ZRI-Report 1/97, Department of Computer Science, University of Kaiserslautern, 1997.

[21] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice-Hall, 1986.

[22] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.

[23] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber, "The design of distributed systems - an introduction to FOCUS," Tech. Rep. TUM-I9203, Technische Universität München, Institut für Informatik, January 1992.

[24] M. Broy, "Semantics of finite and infinite networks of concurrent communicating agents," *DC*, vol. 2, pp. 13–31, 1987.

[25] M. Broy, "Advanced component interface specification," in *Proc. TPPP'94, Lecture Notes in Computer Science 907*, pp. 369–392, 1995.

[26] M. Broy and K. Stølen, "Focus on system development." Book manuscript, January 1997.

[27] C. B. Jones, *Systematic Software Development Using VDM, Second Edition*. Prentice-Hall, 1990.

[28] J. M. Spivey, *Understanding Z, A Specification Language and its Formal Semantics*, vol. 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[29] U. Engberg and M. Nielsen, "A calculus of communicating systems with label-passing," Tech. Rep. DAIMI PB-208, University of Aarhus, 1986.

[30] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part I," *Information and Computation*, vol. 100, pp. 1–40, 1992.

[31] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part II," *Information and Computation*, vol. 100, pp. 41–77, 1992.

[32] R. Grosu, "A formal foundation for concurrent object oriented programming," Phd Thesis TUM-I9444, Institut für Informatik, TU München, Arcisstr. 21, D–80290 München, Germany, Nov. 1994.

[33] M. Broy, "Equations for describing dynamic nets of communicating systems," in *Proc. 5th COMPASS Workshop, Lecture Notes in Computer Science 906*, pp. 170–187, 1995.

[34] R. Grosu and K. Stølen, "A model for mobile point-to-point data-flow networks without channel sharing," in *AMAST'96* (M. Wirsing, ed.), LNCS, 1996.

[35] R. Grosu and K. Stølen, "A denotational model for mobile point-to-point dataflow networks with channel sharing," tech. rep., Technische Universität München, 1996.

[36] G. Boudol, "Asynchrony and the pi-calculus," Research Report 1702, INRIA, 1992.

[37] G. Boudol and G. Berry, "The chemical abstract machine," *Theoretical Computer Science*, vol. 96, pp. 217–248, 1992.

[38] C. Fournet and L. Maranget, *The Join-Calculus language documentation and user's guide*, 1997. Release 1.02.

[39] L. Cardelli and A. Gordon, "Mobile Ambients," 1997.

[40] Inprise Corporation, "Home page, `http://www.inprise.com`," 1998.

[41] F. Huber and B. Schätz, "Rapid prototyping with autofocus," in *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG Fachgespräch 1997, pp. 343-352* (A. Wolisz, I. Schieferdecker, and A. Rennoch, eds.), GMD Verlag (St. Augustin), 1997.

[42] F. Huber, B. Schätz, A. Schmidt, and K. Spies, "Autofocus - a tool for distributed systems specification," in *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems* (B. Jonsson and J. Parrow, eds.), pp. 467–470, LNCS 1135, Springer-Verlag, 1996.

[43] ITU-TS, *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*. Geneva: ITU-TS, September 1993.

[44] UML Group, "Unified Modeling Language," Version 1.1, Rational Software Corporation, Santa Clara, CA-95051, USA, July 1997.

[45] F. Huber, S. Molterer, A. Rausch, B. Schätz, M. Sihling, and O. Slotosch, "Tool supported specification and simulation of distributed systems," in *Proceedings International Symposium on Software Engineering for Parallel and Distributed Systems* (B. Krämer, N. Uchihira, P. Croll, and S. Russo, eds.), pp. 155–164, IEEE Computer Society, Los Alamitos, California, 1998.