Peer-based Automatic Configuration of Pervasive Applications

Marcus Handte, Christian Becker, Kurt Rothermel
Institute of Parallel and Distributed Systems, Universität Stuttgart, Germany
firstname.lastname@informatik.uni-stuttgart.de

Abstract

Pervasive Computing envisions seamless support for user tasks through cooperating devices that are present in an environment. Fluctuating availability of devices, induced by mobility and failures, requires mechanisms and algorithms that allow applications to adapt to changing environmental conditions without user intervention. To ease the development of adaptive applications, we have proposed the peer-based component system PCOM. This system provides fundamental mechanisms to support the automated composition of applications at runtime. In this paper, we discuss the requirements on peer-based automatic configuration of pervasive applications and present an approach based on Distributed Constraint Satisfaction. The resulting algorithm configures applications in the presence of strictly limited resources. To show the feasibility of the approach, we have integrated the algorithm into PCOM and provide an evaluation based on simulation and measurements.

1. Introduction

Pervasive Computing utilizes devices that cooperatively execute distributed applications in order to provide distraction-free support for complex user tasks. In essence, pervasive applications can be seen as compositions of functionality provided by devices in the physical environment of their users. The interaction between applications and users is unobtrusive since many devices become invisible through their integration in everyday objects. The devices encountered in such environments are heterogeneous, ranging form resource-limited specialized systems up to powerful general purpose computers. Due to wireless communication, many devices can be mobile. Hence, the available functionality is continuously fluctuating.

Both, the heterogeneity and the dynamics of the environment increase the complexity that developers, administrators, and users face when they are building, operating, or using applications. While it is possible to shift the responsibilities and thus, the arising complexities, between these parties, e.g., administration requires more fine-tuning but usage becomes simpler, a pure shift is not enough to cope with the complexities.

In response, a number of research projects are focusing on the development of abstractions that enable the automation of various aspects of pervasive systems. One of these aspects is the automatic composition of applications at runtime. This automation is a major rationale behind many pervasive infrastructures, e.g. GAIA [16], AURA [9], and our component system PCOM [3]. At the present time, composition is receiving attention in other research areas as well, e.g., the multimedia [10] and the web services community [15].

In PCOM, the configuration of a component-based application, i.e., the composition of components that constitute an application is automatically determined at runtime. If the resources required by components are limited, finding a single configuration that meets all requirements is an NP-complete problem.

In this paper, we discuss the requirements on peer-based automatic configuration of pervasive applications. Furthermore, we propose an approach towards automatic configuration of PCOM applications based on existing work in the field of Distributed Constraint Satisfaction [18]. In contrast to our previously introduced greedy heuristic [3], the proposed approach is complete. Despite the exponential runtime, our evaluation suggests that a) the approach can be applied to many real-world problems and b) even with limited runtime it can easily outperform the greedy heuristic.

The remainder of this paper is structured as follows. The next section introduces the underlying system model and presents the necessary details of PCOM. Section 3 describes the configuration process and derives the requirements for its automation. The approach is motivated and detailed in Section 4. Section 5 provides an overview of the resulting algorithm that is evaluated in Section 6. Finally, Section 7 describes related work and Section 8 concludes the paper.

2. System Model

As presented in [3] and [4], our work focuses on peer-based pervasive systems. In such systems, devices within communication range connect to each other on-the-fly using wireless communication technology, e.g., Bluetooth or WLAN. Devices offer their functionality and cooperate with other devices in the vicinity in order to execute applications. As a result, applications are composed of functionalities provided by different devices. In contrast to smart environments like GAIA [16], AURA [9] and iROS [14], peer-based systems do not rely on the presence of a centralized coordinating entity. Due to user mobility, the availability of functionalities is continuously fluctuating. As an example consider a group of persons that is cooperatively working with PDAs on a business trip. In this scenario, relying on a central coordinating entity, e.g., a fixed server might prohibit cooperation.

To ease the development of adaptive applications, we have developed the light-weight component system PCOM. In the following, we will sketch the relevant concepts. A detailed description can be found in [3].

PCOM differentiates between components and instances. Components can be thought of as blueprints for their instances. The number of instances is a priori not restricted. Each component resides on exactly one device and each device contains a container that creates and manages all local instances. An instance provides a contractual description of its offered functionality and its requirements. The requirements can be split into local resource requirements and requirements towards other instances. The container assigns the local resources and in cooperation with other containers it starts and manages the required instances. An application is a tree of component instances that is constructed by recursively starting the required instances of a root instance, the so-called application anchor. An application can only be started, if all required instances can be executed. A PCOM container must be able to manage strictly limited resources, e.g., to model exclusive resources like input devices, etc. Usually, introducing limited resources will lead to a limitation on the number of instances that can be executed. Since instances of different components can have overlapping resource requirements, e.g., both require some memory, starting an instance of one component can prohibit the instantiation of another one.

3. Automatic Configuration

Automatic configuration denotes the task of automatically determining a composition of components that can be instantiated simultaneously as application. Such a composition is subject to two classes of constraints. The first class are structural constraints. They describe what constitutes a valid composition in terms of functionalities. The second class are resource constraints. They are a result of the limited resources.

Structural constraints can be either specified in advance, e.g., as an architectural model expressed in some description language, or they can be individually associated with components, e.g., as recursively specified dependencies contained in contracts. If an architectural model is available, the configuration must ensure the availability of a matching instance for each modeled component. If structural constraints are specified per component, the configuration must ensure that all recursively required instances are available.

Resource constraints can be modeled in various ways. For the sake of simplicity, this paper relies on a simple but powerful model that is also used in [20]. Each instance specifies its local resource requirements in advance as an integer vector where each index denotes a specific resource and the integer denotes the required amount. The vector might vary depending on the usage of the instance. Similarly, the available resources on each device can be modeled as a vector. Since the availability of resources can change, the values might fluctuate at runtime. To satisfy the resource constraints, the configuration has to ensure that at any time the index-wise sum of all requirement vectors of local instances is index-wise less than or equal to the vector that specifies the locally available resources.

The complexity of automatic configuration arises from the fact that both, resource and structural constraints must be fulfilled simultaneously. Due to the recursive definition of structural constraints in PCOM, it is not possible to calculate the resource requirements of a certain sub tree in advance without determining all possible configurations of that sub tree. But even if it was possible, the strictly limited resource availability might lead to exclusions between structurally possible configurations of arbitrary sub-trees. Note that in general finding all exclusions is as complex as finding a configuration.

3.1. Example

In the following, we will briefly describe the process of automatic configuration based on PCOM using an exemplary application. Figure 1 depicts an environment that consists of three devices. Each device has a certain amount of resources. The PDA has a single display (DSP), a certain amount of memory (MEM) and CPU. Each device hosts some components. The laptop hosts a component that enables a remote system to access the file system (File System) and another one that is capable of displaying a presentation (Remote PPT). Each instance of this component requires CPU, memory and access to the local presentation library (PLIB). Furthermore, an instance of this component requires two displays.

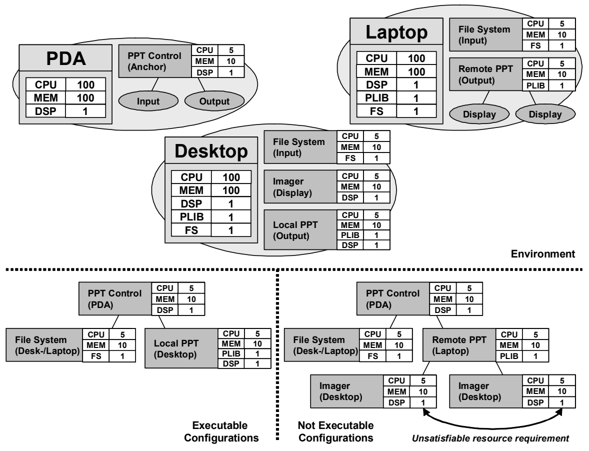

Figure 1. Environment and Configurations

If an instance of the application anchor (PPT control) is started, the container on the PDA must assign the resources and it must resolve the dependencies (Input and Output). In this example, Input can be resolved using an instance of the File System component on the laptop or on the desktop. Output can be resolved by Remote PPT on the laptop or Local PPT on the desktop. If the PDA uses the Remote PPT, the laptop must assign the resources and resolve the displays. In this environment, there are four structural possibilities to configure the application depending on the choice for Input and Output (c.f. Figure 1). Since the Imager can only be started once due to the limitation of DSP resources, there is no way of using Remote PPT in such a way that all constraints are met. The two executable configurations use a File System on the desktop or on the laptop and the Local PPT. This example also demonstrates the interrelation of resource and structural constraints. Choosing an instance that represents a locally valid option can still lead to unsatisfiable requirements that can only be discovered gradually.

3.2. Requirements

The requirements towards peer-based automatic configuration can be derived directly from the presented system model and the overall vision of Pervasive Computing with respect to the distraction-free support of user tasks:

Completeness: If a valid configuration exists automatic configuration should be able to determine one. Also, it should be capable of detecting that a certain application is currently not executable at all. Otherwise, users might eventually become frustrated. However, since the problem of finding a single configuration is NP-complete, achieving completeness for arbitrary problem instances is not practicable. Thus, in practice we can only demand that automatic configuration is capable of finding solutions in a broad range of different environments. As we will discuss in the evaluation section, a complete approach whose runtime is limited is often preferable over a heuristic that "arbitrarily" ignores a number of possible solutions.

Efficiency: As the configuration delay of complete solutions for automatic configuration will increase exponentially with the size of the problem, efficiency becomes a major requirement. Since long configuration delays might lead to frustrated users, automatic configuration should include as many optimizations as possible to enable speed-ups without overloading resources or sacrificing completeness.

Optimism: Ideally, an algorithm for automatic configuration should be fast in resource-poor as well as resource-rich environments. Typically there is a tradeoff between optimizing worst- and best-case scenarios. Since users would expect to achieve speedups by adding resources, optimizations of the worst-case delays at the cost of higher execution times in resource-rich environments are not desirable. Therefore, automatic configuration should be optimistic.

Distribution: In peer-based systems the availability of a powerful and reliable device cannot be guaranteed. As a result, the scalability of a centralized approach will be limited in environments that consist of a large number of resource-poor devices. In order to utilize the inherent parallelism and the resources of such environments, automatic configuration should be performed cooperatively by the available devices.

Resilience: The mobility of users and devices in pervasive systems leads to continuous and possibly unpredictable fluctuations regarding the availability of functionalities. As a result, applications in such systems have to cope with the resulting dynamics at runtime. Since an algorithm for automatic configuration might be running a couple of seconds, the algorithm itself should be capable of dealing with fluctuations that can be detected during its execution.

4. Approach

In general, finding a single executable configuration in the presence of strictly limited resources is an NP-complete problem. This can be shown, for instance by interpreting a conjunctive normal form that is known to be NP-complete for more than three literals (3-SAT) [5] as components with specifically constructed resource constraints. Thus, approaches for automatic configuration can apply NP-complete formalisms.

As we will show in the following section, automatic configuration can be mapped to a Constraint Satisfaction Problem. Informally, Constraint Satisfaction Problems can be described as follows: Given a set of variables with finite domains and a set of constraints between variables, find a valid variable assignment such that all constraints between the variables are met.

Due to the specifics of the peer-based system model, centralized approaches towards solving Constraint Satisfaction Problems cannot fulfill the requirement regarding distribution. The foundations for distributed algorithms have been developed in the field of Distributed Artificial Intelligence. In this field, the notion of Distributed Constraint Satisfaction Problems has been formalized [18]. There, the set of variables and constraints is distributed across a number of agents. Each agent is responsible for assigning its variables and evaluating its constraints. An overview and a classification of distributed algorithms for solving such problems can be found in [19].

From this set of algorithms, we show how Asynchronous Backtracking (ABT) [18] can be extended to fulfill all requirements towards peer-based automatic configuration. ABT is a sound and complete dependency-directed backtracking algorithm. It enables agents to concurrently assign values to their variables and thus has the potential to use the available parallelism. As we will show later on, it is possible to construct a mapping that enables the algorithm to start processing without any further distribution of knowledge. In the best case, it simply assigns all variables the right value and terminates. In contrast to consistency algorithms that first try to eliminate some illegal options before they assign values to variables, this algorithm fulfils the requirement towards optimism. Apart from that, the dependency-direction of the algorithm reduces the search within irrelevant possibilities by only considering options during backtracking that have the potential to resolve the conflict. This has the potential to greatly increase the efficiency of automatic configuration in many environments. Finally, as we will discuss later on, an extension to achieve resilience can be added in a straight-forward manor.

4.1. Configuration as Constraint Satisfaction

To use ABT as basis for automatic configuration, the functionalities present in an environment as well as structural and resource constraints must be represented as variables, domains and constraints between variables. To model PCOM applications, we map dependencies to variables, components to domains and the structure with resource requirements to constraints.

To model structural constraints, each component instance is represented as a multi-dimensional variable where each dimension denotes a dependency towards another instance. The domain of each dimension is given by the available options for the corresponding dependency. For the PPT Control (c.f. Figure 1) that has two dependencies Input and Output, we create a two dimensional variable. If there are two possibilities to satisfy the dependency Output (Remote and Local PPT) and one for the dependency Input (File System), the domain of the variable will be [0,1],[0]. Note that the domain solely contains direct possibilities. Due to the recursive nature of dependencies, there might be many possibilities to configure each of the assignments, e.g., if there were multiple Imagers, there would be multiple ways to configure a Remote PPT.

A difference between constraint satisfaction and automatic configuration is that constraint satisfaction determines an assignment for all variables. Automatic configuration determines a partial solution that satisfies the constraints, i.e. if an instance is not required, the dependencies of this instance must not be resolved. To model this, the domain of each dimension is extended with the pseudo value $\varnothing$. Thus, the domain for the previous example would be [$\varnothing$,0,1],[$\varnothing$,0]. One can think of the dependencies whose component has not been discovered and used as set to $\varnothing$. This effectively transforms the search for a partial solution in a search for a complete solution.

Now that the variables and domains are defined, the mapping must ensure that only structurally valid configurations are generated (c.f. Figure 2). This can be achieved by two constraints. Both can be motivated by looking at the Output dependency. There are two possible instances (Remote and Local PPT) that can be selected to fulfill this dependency. Since the configuration requires only one at a time, we can add the constraints that Remote PPT needs to be considered iff its parent instance assigns values that contain 0 in the first dimension. Similarly, Local PPT needs to be considered iff 1 is assigned to the first dimension of the variable. Furthermore, another constraint is required that ensures that the pseudo value is used iff the component instance is not used by its parent. Apart from these recursively defined constraints, one additional constraint must ensure that the anchor is always instantiated. Otherwise, the trivial configuration that contains only a non-instantiated anchor also fulfils all structural requirements. Finally, the configuration must consider the resource constraints on each container. Thus, we add a constraint to ensure that the resource consumption of all instances that are executed on the container must not exceed its available resources.

Let *VAL(c, n)* be defined as the current assignment for the variable dimension *n* of component instance *c*.

Each anchor instance *a* with *m* dependencies is subject to:

(1) $\forall n \in \{1 \ldots m\}$ $VAL(a, n) \neq \varnothing$ *(anchors must be resolved)*

Each non-anchor instance *c* with m dependencies that is referenced by dependency *j* under the value assignment of *k* of its parent instance *p* is subject to:

(2) If $VAL(p, j) \neq k$: $\forall n \in \{1 \ldots m\}$ $VAL(c, n) = \varnothing$ *(unused instances are not resolved)*
(3) If $VAL(p, j) = k$: $\forall n \in \{1 \ldots m\}$ $VAL(c, n) \neq \varnothing$ *(used instances are resolved)*

Each container *z* that has the resources *r* and hosts the instances $c_1, \ldots, c_n$ that require the resources $r_1, \ldots, r_n$ is subject to:

(4) $r >= \Sigma_i (r_i)$ *with* $i \in \{1 \ldots n\}$ *(resource requirements are met)*

Figure 2. Configuration Constraints

To guarantee termination, ABT requires a total priority ordering between variables to create a cycle-free constraint network. Note that this ordering also defines the strategy for resolving conflicts during backtracking. A partial ordering is introduced by the structural constraints of applications, i.e., each child instance must have a lower priority than its parent. The remaining degree of freedom can be filled by some arbitrary ordering scheme. However, in order to be usable, ABT must be able to create the scheme gradually. Otherwise the search space would have to be unfolded upfront which conflicts with the requirement of optimism.

To demonstrate this, consider the search space shown in Figure 3 (a) that consists of components A-F, with the dependencies 1-4, and containers C1 and C2 where A, B, E reside on container C1 and C, D, F reside on container C2. The dashed lines indicate resource constraints and the solid lines indicate structural constraints. Note that the structural exclusion constraint between C and D is implicit since A will only assign one value at a time for its dependency 2. Also note that the dependencies and containers can be thought of as variables of the CSP.

Since A is the only component that is known a priori and the others must be discoverable in parallel, we can only introduce a local ordering as shown in Figure 3 (b) by assigning locally unique ids to dependencies and their possible options. From these local ids, we can create a global id by concatenating the ids along the path (e.g, 1,2,1,1,1 for E or 1,2,2 for D). On these ids, we can now define comparison operators to establish a total ordering. In order to adhere to the partial ordering introduced by the structural constraints, we must ensure that all ids that are totally included as a prefix in another id have higher priority, i.e., 1,2,1,1,1 < 1,2,1. Apart from that we can for instance either decide that the length of an identifier is first compared and the longer the identifier, the lower the priority and for identifiers with equal length, we use their values for comparison. Another possible option would be to compare the values first before comparing the length.

That way an ordering can be established that would lead to a backtracking strategy where lower levels would be reconfigured before higher levels are. Alternatively one could define an ordering where backtracking would take place in one subtree before it moves to the conflicting component of another subtree.

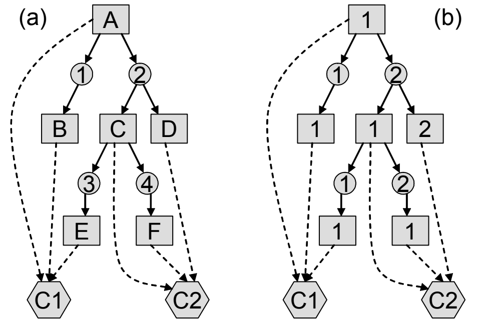

Figure 3. Numbering Scheme Example

Figure 4 shows such orderings. Note that in order for this ordering to work, every component needs to know its place within the tree, i.e., the concatenated id. Thus upon its first usage, each component needs to be supplied with the identifier that its parent assigns for it.

For our implementation we have chosen the strategy that reconfigures components on lower levels first. The idea is that lower level components have less recursively required instances that must be notified if their parent changes and thus, reduces the communication overhead. However, this is a heuristic and there might be cases where this might lead to higher overhead.

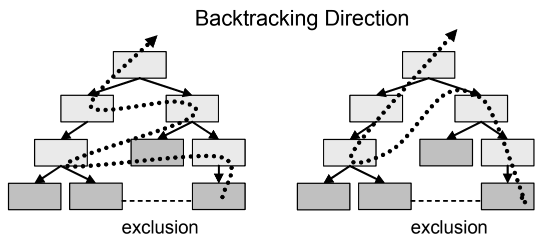

Figure 4. Traversal Strategies

As stated in section 3.2, automatic configuration must be capable of dealing with fluctuations that occur during its execution. Such fluctuations might be the result of the unavailability of local resources or remote devices. In pervasive systems, these fluctuations are typically hard to predict. Consider for instance a user that removes a USB device from a laptop or a traveling user that carries a number of devices. Fortunately, failure-handling for both types of fluctuations can be added in a relatively straight-forward way. Whenever the unavailability of a device is detected, the algorithm on every remaining device simply creates an additional constraint for every instance that has been used on the unavailable device. These new constraints state that these instances can never be used. Since ABT does not impose any timing constraints on the reception of constraints, they can be added without further precautions. Similarly, if a required resource becomes unavailable, the corresponding constraints must be added. The new constraints will eventually lead to a reconfiguration or an unsuccessful termination of the algorithm.

ABT terminates unsuccessfully if an empty constraint set is generated during the execution, i.e., if there is no further choice that can be reconsidered in order to resolve an unsatisfied constraint. Due to the tree structure of applications, such an empty set can only be generated by the anchor. All other component instances can always ask their parents to reconfigure themselves in such a way that they are no longer used. Thus, an unsuccessful run will be recognized by the anchor. The successful termination of the algorithm is achieved if all participating devices stopped generating new messages and all messages have been delivered and processed. Therefore, detecting the successful termination is an instance of a Distributed Termination Problem. As the termination protocol must be resilient to mobility, a simple protocol as described in [6] is not enough. Although, our current implementation does not incorporate such a protocol, its addition as described in [12] should be possible.

Clearly, due to the unpredictable nature of pervasive systems, no termination protocol can guarantee that a successful termination of the algorithm will allow a successful application start up. If a resource becomes unavailable at exactly the same instant of time when the successful termination is detected, there is nothing that can be done. At the present time, the only approach that we can propose is to start the configuration process all over again if such a situation occurs. Another possibility is to start the partial configuration and determine possible adaptations. This, however, is subject of our current research and a discussion lies beyond the scope of this paper.

5. Algorithm

In the following, we provide an overview of the resulting algorithm and we describe some interesting details of our implementation. For the sake of clarity, the pseudo code of the algorithm (c.f. Figure 5, 6) does not consider different applications. Also, the algorithm does not contain optimizations, e.g. only sending messages to containers that require it. The layout borrows from the description of ABT [18]. Note that we need to extend the algorithm with the capability of hosting multiple instances and the resource validation procedure. Furthermore, in order to support the dynamic discovery of components, we add a method that performs discovery and initializes the variables. In this paper, the algorithm is modeled as a reactive process that responds to incoming messages (receive_XXX procedures). Our PCOM implementation allows batch processing of messages.

```
receive_update(identifier, component, value)
  // config denotes the local knowledge about an instance
  config = getConfig(identifier)
  // this happens if the instance is selected for the first time
  if (! config exists)
    // here the variables and their domains are determined
    config = createConfig(identifier,component)
  // this adds the variable assignment to the local knowledge
  config.add(value)
  // finally, all consistency checks are performed
  check_constraints(config)

receive_backtrack(identifier, conflicts)
  // determine whether the conflicts are still conflicting
  if (! conflicts outdated)
    // retrieve the addressed conflict
    config = getConfig(identifier)
    // add the conflicts as a new constraint
    config.addConstraint(conflicts)
    for each id in conflicts
      if (! connected id)
        // create links to keep informed about changed values
        create link between parent of id and config
        // add the value of the conflict to the local knowledge
        config.add(identifier)
      // temporarily copy the currently selected components
      copy = config.getAssignment()
      // perform the consistency checks
      check_constraints(config)
      if (copy == config.getAssignment())
        // if the values have been consistent also send updates
        send_update(identifier, copy) across links

check_constraints(config)
  // if there are unmatched constraints
  if (! config.isConsistent())
    // determine whether a valid assignment can be found
    if (! config.assignConsistent())
      // if not, start or continue backtracking
      backtrack(config)
    else if (reserve_resources(config))
       // else determine whether local resource constraints are met
      assignment = config.getAssignment()
      // if they are met, send the updated assignment
      send_update(identifier, assignment) across links

backtrack(config)
  if (config.isAnchor())
    terminate unsuccessfully
  else
    // determine conflicting instances sets
    conflict_sets = minimum conflict sets
    for each conflicts in conflict_sets
      // send a backtrack message to the lowest instance
      id = minimum identifier in conflicts
      // this message could be remote or local
      send_backtrack(id,  conflicts)
      // remove the conflicting assignment
      config.remove(id)
   check_constraints(config)
```

Figure 5. Algorithm (1)

Since each container can host multiple component instances, the algorithm must be capable of uniquely identifying them. To globally identify an instance and its position within the application, our implementation uses the generated ID discussed in Section 4.1. The application anchor has the ID {}, the first instance for the first dependency of the anchor is identified by {(0)[0]}. The second instance for this dependency is identified by {(0)[1]}. Thus, IDs are arbitrarily long sequences of pairs, where the first index of a pair denotes the dependency and the second index denotes the instance used to satisfy this dependency.

```
reserve_resources(config)
 // if the instance is selected by its parent
 if (config.isInstantiated())
   // and the resources are not reserved
   if (! config.isReserved())
     // try to reserve the required resources
     if (reserve resources for config)
       config.setReserved(true)
       // if the reservation succeeds, continue
       return true
     else
       // if the reservation fails determine conflict sets
       conflict_sets = minimum conflict sets
       // a flag that indicates whether the conflict has been resolved
       reservable = true
       for each conflicts in conflict_sets
         // pick the instance with the lowest identifier
         id = minimum identifier in conflicts
         // backtrack to the parent of the lowest instance
         send backtrack to parent of id with conflicts
         // deactivate the instance that caused backtracking
         c = getConfig(id)
         c.remove(id)
         check_constraints(c)
         // determine whether the current instance is a conflict cause
         if (config.getIdentifier() == id)
           // if it is, it will be uninstanciated after the backtracking
           reservable = false
       if (reservable)
         // the cause of all conflicts has been removed
         reserve resources for config
         config.setReserved(true)
         return true
       else
         // the instance has already been deactivated
         return false
 else
   // if the instance is not used by the parent
   if (config.isReserved())
     // remove the resource reservation
     remove reservation for config
     config.setReserved(false)
   return true
```

Figure 6. Algorithm (2)

To describe the algorithm, we will use the presentation application example introduced earlier (c.f. Figure 7). When the user starts the application, the container calls the receive_update procedure with the ID {}, an identifier that locally identifies the PPT Control component and the value {}. This signals that an anchor should be started (a). Since this is the first time that the configuration algorithm sees an update for {}, it creates a configuration object for this instance. Using the contract of the instance, it determines that PPT Control has two dependencies, thus it creates a two-dimensional variable [Input],[Output]. To determine the domain of the variable, i.e. possible options to satisfy the dependencies, the container performs local and remote lookups (b). Thereby, the algorithm discovers the following options: File System (desktop) {(0)[0]}, File System (laptop) {(0)[1]}, Remote PPT (laptop) {(1)[0]} and Local PPT (desktop) {(1)[1]}. Thus, the domain is [∅,0,1],[∅,0,1]. For the new variable, the initial assignment is [∅],[∅].

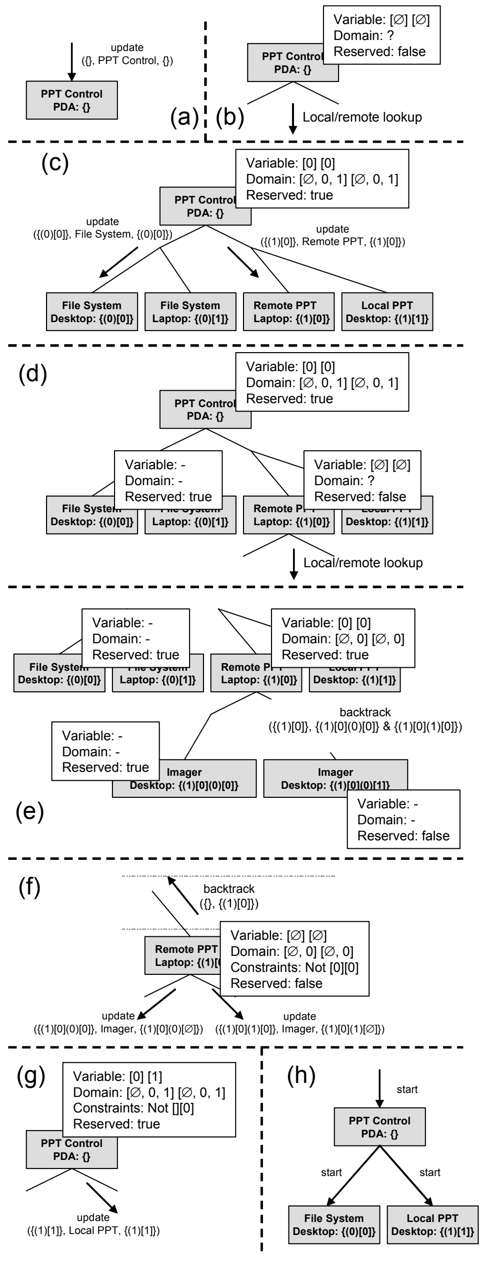

Figure 7. Example

The algorithm continues to add the value {} to the local knowledge which states that the instance bound to the configuration object is instantiated. Thereafter, the algorithm calls the check_constraints procedure and determines that the current assignment [∅],[∅] is not valid, since the instance is used according to the local knowledge. Note that this is a result of the built-in constraints shown in Figure 2. Next, the algorithm determines a valid assignment [0],[0] and reserves the resources using the reserve_resources procedure. The reservation finishes successfully and the algorithm continues to send parallel update messages to the File System {(0)[0]} and the Remote PPT {(1)[0]} (c).

When the update message for the File System arrives, the algorithm creates the configuration object, adds the value to the local knowledge, performs the resource reservation, and stops without sending further messages (d). In response to the update for the Remote PPT, the algorithm sends two updates to the Imager ({(1)[0](0)[0]} and {(1)[0](1)[0]}). The first update message creates a new configuration object and finishes successfully. The second update fails due to a lack of resources. Thus, the reserve_resources procedure determines that the minimum conflicting sets consist of exactly one set of component instances that contains both instances of the Imager component (e).

Note that although the File System is also running on the desktop, its identifier will not be added to the conflict set since it has nothing to do with the shortage on displays. Furthermore, the algorithm does not need to add the complete path to the anchor to the constraint as it can be gradually generated whenever a conflict is escalated. Following the traversal strategy, {(1)[0](1)[0]} is picked as the smallest identifier and a backtrack message is sent to is parent. Additionally, the instance is deactivated and all potentially reserved resources and required instances are released by calling check_constraints.

When the backtracking message arrives at the Remote PPT, the component will determine whether it has to create any new links. Since both identifiers contained in the conflict set are local variables, no new link must be created. Therefore, the algorithm continues to add a mutual exclusion constraint between {(1)[0](0)[0]} and {(1)[0](1)[0]} to the local knowledge. In cases where added conflicts are not conflicts between linked instances, the addition of new links between the assigning instance and the instance that recorded the constraint are necessary to ensure that the constraint evaluation always considers all relevant variable assignments of the present situation.

Since the Remote PPT cannot create a valid assignment, it creates a backtracking message that contains its own identifier and sends it to its parent. Thereafter, the Remote PPT is deactivated and its constraints are checked again. Thereby, the algorithm releases all resources, assigns [∅],[∅] and creates updates that will eventually release previously bound instances (f). When the PPT Control receives the backtracking message, it adds the constraint that the Remote PPT can never be started and assigns another value for the Output dependency. It selects the Local PPT {(1)[1]} and it creates an update (g). When the update arrives, the Local PPT will be reserved and the algorithm stops.

If the Laptop becomes unavailable, the algorithm simply creates backtracking messages for each used component instance provided by the laptop. These backtracking messages solely contain the identifier. The same procedure is performed, if some reserved resource becomes unavailable. The algorithm determines the conflicting sets and creates the corresponding backtracking messages.

A termination protocol can be added by wrapping the receive_XXX procedures and the send statements. The necessary steps that need to be performed depend on the chosen protocol. A simple protocol for double counting messages would increase the send and receive counters whose state is later on compared by a wave of termination messages that is sent along the tree structure. A resilient termination protocol would have to perform further steps (see [12] for details).

When the algorithm succeeds, the application must still be started. Therefore, an asynchronous traversal of the tree-structure starting from the application anchor is sufficient. This will not result in conflicts, since each configuration object has already reserved the resources for the chosen bindings (h).

Since the algorithm above is a modified instance of ABT, the proof of correctness follows the argumentation provided in [18]. Due to space restrictions, we would like to refer the reader to this paper for further details. However, an interesting difference between ABT in general and the special instance discussed in this paper is that the application anchor can detect an over-constrained environment due to the tree-structure of applications.

6. Evaluation

As discussed in Section 4, ABT fulfills the requirements regarding completeness, optimism, distribution and resilience. In this section, we discuss efficiency as the last remaining requirement.

ABT resolves unrelated conflicts simultaneously and it reconsiders only those instances that have the potential to resolve a conflict. Thus, the configuration complexity depends on the induced width, i.e. the size of sub problems that can be solved independently, and not the total width of the search space [2]. The induced width of automatic configuration depends on the number of structurally valid configurations and the locality of resource conflicts, i.e. the number of instances that have conflicting requirements towards the same resources. In many pervasive systems, resource conflicts can be assumed to be relatively local. To justify this, consider that the worst-case runtime occurs, if many instances are executed on one device and a widely used resource (e.g. memory or CPU) is not available. However, the integration of devices into everyday objects leads to environments where the majority of devices are specialized embedded systems. Just like everyday objects, they will be tailored towards a small number of specific functionalities, which will increase the locality.

The number of structurally valid configurations depends on the number of available components that can be used within the application and thus it heavily depends on the capabilities of the environment. To analyze the effects of an increasing number of possibilities we ran a number of simulations with a discrete event simulator. Within one time step, the simulator processes all messages that have been sent and creates all new messages before it moves on to the next time step.

The simulated environments have been constructed using the following procedure: we create an application that consists of n instances by adding n components to a binary tree from left to right, top to bottom. Then we create one container and place the anchor on it. For the remaining (n-1) components we create m containers and place them on the containers round-robin. Thereby, we set the resource requirements of each component to one unit of one resource that is used by all components on the container. Furthermore, we set the available amount of the resource to the number of components that are hosted on this container. Then we randomly pick k components and replicate them on randomly selected containers. Hereby, we set their resource requirements for the commonly used resource on that container to two without increasing the resources on this container. Thus, increasing k will lead to a higher potential for conflicting selections during automatic configuration and decreasing the number of containers m will decrease the locality of the resulting conflicts.

Note that there will always be exactly one configuration that can be started which consists solely of instances provided by the initially placed components. The exact amount of messages might vary depending on the arrival time of messages. For instance, if a device requires a long time to detect or propagate a local conflict, the number of messages as well as the required duration might increase or decrease depending on the scenario.

Figure 8 depicts simulation results in cases where the locality of conflicts is high, i.e. m = (n-1) / 2, for different application sizes (n = 8, 12, 17) and a different number of conflicting components (k = 0 to 30). Each measurement shows the average, respectively the maximum, of 100 runs.

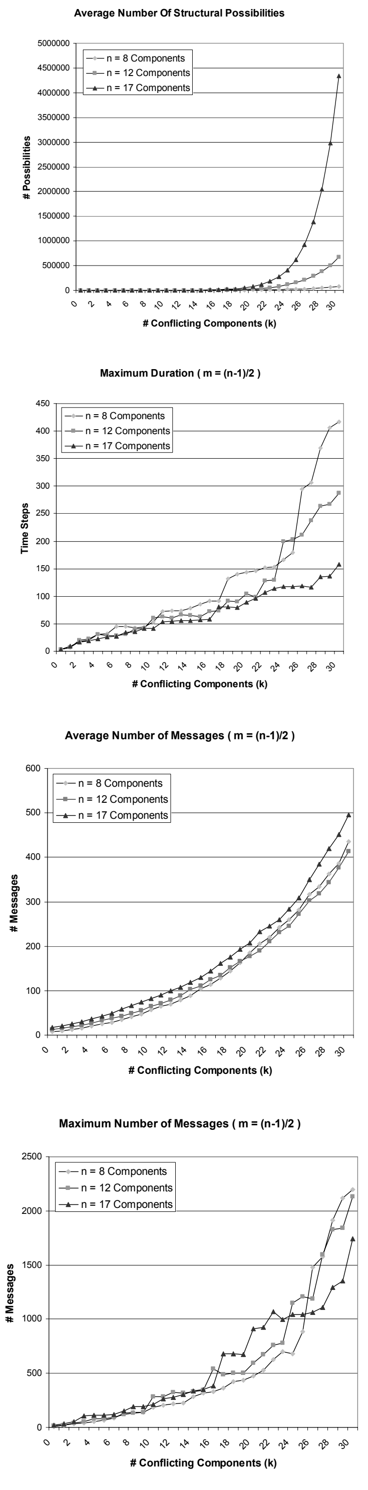

Figure 8. Simulation Results with Locality

The simulations show that the number of messages required to determine configurations grows exponentially. For k ~ 17 the maximum number of messages exceeds 400. This is a result of the exponential increase of structural possibilities for configurations which in turn can be attributed to the (exponential) way conflicts are created, i.e. by cloning components. Note that the number of messages does not necessarily lead to a high configuration delay as the solution is found in less than 90 time steps.

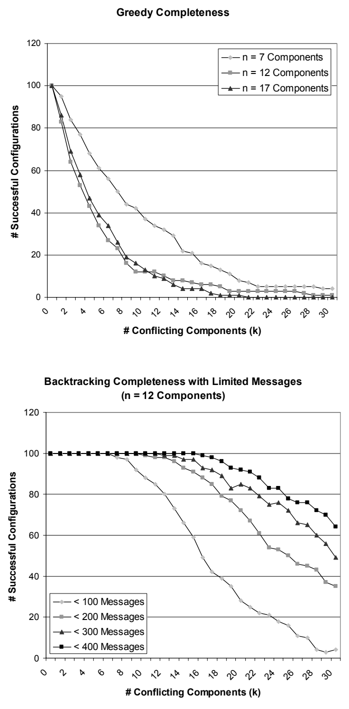

Figure 9. Completeness with Locality

One might argue that the worst-case message overhead prohibits the application of the algorithm. Therefore, we have compared the achievable completeness if the number of messages is limited to 100, 200, 300 and 400 with the completeness that can be gained from a greedy heuristic that selects sub trees recursively without ever reconsidering a choice as proposed in [3]. Figure 9 shows the success rates for an application with 12 instances. In average, the heuristic produced 23-100 messages, but for k = 15, it can only find the configuration in 8 cases whereas backtracking finds 59 with 100 and all with 400 messages. Thus, even if the complete algorithm would have been manually aborted, the success rate would have been higher.

If we construct a scenario where there is no valid configuration by increasing the resource requirements of one initially placed component by one, the number of transferred messages increases by approximately a factor of two. This can be attributed to the min-conflict value ordering heuristic that is used to select instances. However, in over-constrained search spaces aborting the process does not affect completeness.

Finally, Figure 10 shows the success rate in a case where the locality assumption of conflicts does not hold. Instead of increasing the number of containers as the size of the application grows, we fix the number to 4. Despite the increasing message overhead, the complete algorithm is still able to outperform the greedy heuristic in terms of completeness.

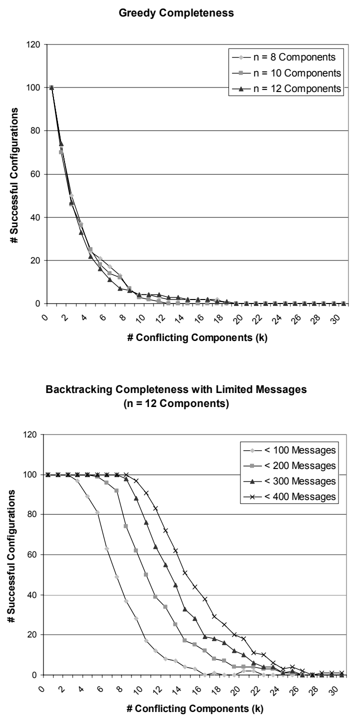

Figure 10. Completeness without Locality

To determine the configuration delays in a real system, we have implemented a prototypical version of the algorithm as part of PCOM. To provide values for small devices, we placed an application with 7 components on 2 Pocket PCs (XScale 400MHz / 10 MBit WLAN) using the procedure described above. Since all components were using the same resource on each of the Pocket PCs, this experiment reflects a situation where the locality of conflicts is low. We ran 7 scenarios with 0, 2, 4, 6, 8, 10 and 12 randomly created conflicting components. For each number of conflicting components we performed 10 measurements. Figure 9 shows the results of these measurements with respect to configuration delay (including distributed termination detection using a simple credit-based protocol and application startup), number of local and remote messages created by the greedy and the backtracking algorithm as well as the achievable completeness of the backtracking algorithm within bounded delays.

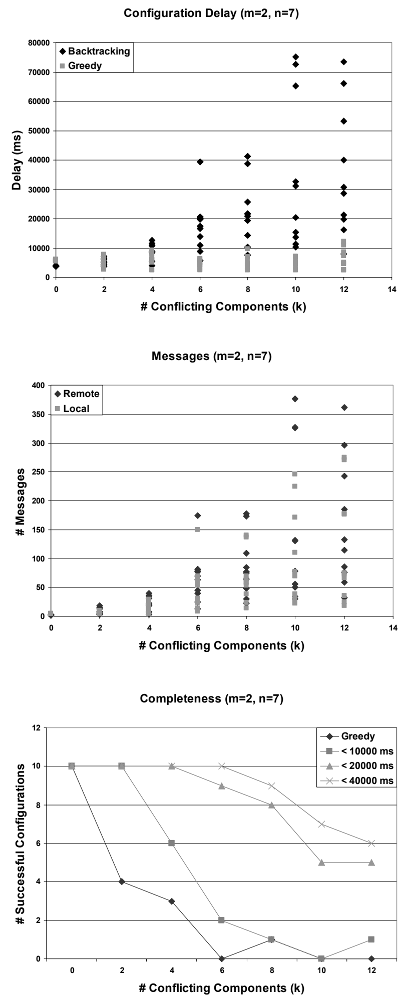

Figure 11. Measured Results without Locality

The measurements indicate that there is a high correlation between the number of remote messages and the configuration delay. This can be attributed to the fact that the locality of conflicts is low. However, the completeness that can be achieved with bounded delay is always higher, even if the delay is limited to 10 seconds (only slightly higher than the average runtime of the greedy algorithm with ~8-9 seconds).

7. Related Work

As most projects in Pervasive Computing deal with distributed functionalities, they have to address the management of compositions. The degree of automation varies heavily depending on the focused system model. The GAIA project [16] for instance separates the implementation of functionalities from the composition of applications using two externalized mappings. Since GAIA assumes a partially static environment, it is not necessary to automate these mappings. The AURA project [9] uses a task abstraction that is mapped onto functionalities available in a certain environment. The mapping is done by a centralized environment manager that coordinates the functionalities of its environment. In contrast to PCOM and GAIA, functionalities in AURA are self-contained entities that solely interact implicitly through users. The iROS [14] system provides a generic mechanism that enables interaction between functionalities. Since iRos does not impose constraints on the available components, the management of the composition must be performed manually. Similarly, One.World [8] does not support the automated management of compositions. Instead, the system shifts this responsibility to the developer.

The Pebbles project [17] uses an abstraction called goal to model an application and it uses a planning engine that automates the creation of valid configurations at runtime. To the best of our knowledge Pebbles uses a centralized planning engine. Another system that uses centralized planning to configure component-based applications is Planit [1]. Planit uses temporal refinement planning to adapt and configure applications at runtime. In contrast to the proposed approach, centralized approaches require a global view of the components and resources of the environment.

Automatic configuration as discussed in this paper can be seen as instance of a distributed resource allocation problem. In the past, research has been applied to distinct domains, e.g. job scheduling [11] or patient scheduling [7]. However, these domains are different from automatic configuration since they try to allocate a set of tasks that is known in advance. In the discussed approach the set of components is discovered at runtime. This requires that the set of variables and domains can be gradually created from the discovered components.

More recently, researchers developed the notion of Dynamic Distributed Constraint Satisfaction Problems, e.g. to perform distributed monitoring in sensor networks [13]. To deal with the dynamics of the environment, constraints need to be added or removed depending on a predicate. This is similar to the required extension for resilience as all constraints depend on a predicate that continuously evaluates the availability of devices and resources. However, the approach presented in [13] does not deal with discovery.

8. Conclusion

In this paper, we have discussed the requirements on automatic configuration in peer-based pervasive systems. Furthermore, we have presented a mapping that enables the automatic configuration of component-based applications in PCOM using Distributed Constraint Satisfaction techniques. The feasibility of this approach has been evaluated using simulation and a prototypical implementation of the algorithm. The results indicate that the presented complete approach is preferable over the greedy heuristic. Although it is possible to construct scenarios in which the complete algorithm will have an unacceptable delay, we are confident that many real-world problems will exhibit the locality to keep the delay within acceptable bounds.

In the near future, we will extend the presented work towards runtime adaptation where the cost for reconfiguring an executed partial application must be taken into account. Also, we are planning to investigate hybrid systems that might contain coordinating entities at certain times. In such systems, a fragment of the state of the environment could be collected at each of the available coordinators which in turn could thereafter cooperatively configure applications.

9. Acknowledgements

We would like to thank the reviewers for their valuable suggestions that helped us improving the paper and removing ambiguities. Furthermore, we would like to thank our colleagues Gregor Schiele and Pedro Marron for their comments and support during the early stages of this paper.

This work is funded by the German Research Foundation within DFG Priority Programme 1140 – Middleware for Self-organizing Infrastructures in Networked Mobile Systems.

10. References

[1] Arshad, N., Heimbigner, D., Wolf, A.: Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems, 15th IEEE Intl' Conference on Tools with Artificial Intelligence, pp. 39-47, 2003

[2] Baker, A.: Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Empirical Results, PhD Thesis, University of Oregon, 1995

[3] Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM – A Component System for Pervasive Computing, 2nd Intl' Conference on Pervasive Computing and Communication, pp. 67-77, 2004

[4] Becker, C., Schiele, G., Gubbels, H., Rothermel, K.: BASE – A Micro-broker-based Middleware for Pervasive Computing, 1st IEEE Intl' Conference on Pervasive Computing and Communication, pp. 443-451, 2003

[5] Cook, S.: The complexity of theorem-proving procedures, Proceedings of the 3rd Annual Symposium on Theory of Computing, pp. 151-158, 1971

[6] Dijkstra, E., Scholten, C.: Termination Detection for Diffusing Computations, Information Processing Letters, vol. 1, no. 11, 1980

[7] Decker, K., Li, J.: Coordinated Hospital Patient Scheduling, 3rd Intl' Conference on Multi-Agent Systems, pp. 104-111, 1998

[8] Grimm, R.: One.World: Experiences With a Pervasive Computing Infrastructure, IEEE Pervasive Computing, vol. 3, no. 3, pp. 22-30, Jul.-Sept. 2004

[9] Garlan, D., Siewiorek, D., Smailagic, A., Steenkiste, P.: Project Aura: Towards Distraction-Free Pervasive Computing, IEEE Pervasive Computing, vol. 1, no. 2, pp. 22-31, Apr.-Jun. 2002

[10] Gu, X., Nahrstedt, K., Chang, R., Ward, C.,: QoS-Assured Service Composition in Managed Service Overlay Networks, 23rd IEEE Intl' Conference on Distributed Computing Systems, pp. 194-204 , 2003

[11] Liu, J-S., Sycara, K.: Multiagent Coordination in Tightly Coupled Task Scheduling, 1996 Intl' Conference on Multi-Agent Systems, 1996

[12] Lai, T., Wu, L.: An (N-1)-Resilient Algorithm for Distributed Termination Detection, IEEE Transactions on Parallel and Distributed Systems, vol. 6, no. 1, pp. 63-78, 1995

[13] Modi, P., Jung, H., Tambe, M., Shen, W-M., Kulkarni, S.: A Dynamic Distributed Constraint Satisfaction Approach to Resource Allocation, 7th Intl' Conference on Principles and Practice of Constraint Programming, pp. 685-700, 2001

[14] Ponnekanti, S., Johanson, B., Kiciman, E., Fox, A.: Portability, Extensibility and Robustness in iRos, 1st IEEE Intl' Conference on Pervasive Computing and Communications, pp. 11-20, 2003

[15] Raman, B., Katz, R.H.: An Architecture for Highly Available Wide-Area Service Composition, Computer Communication Journal, vol. 26, no. 15, pp. 1727-1740, 2003

[16] Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R., Nahrstedt, K.: A Middleware Infrastructure for Active Spaces, IEEE Pervasive Computing, vol. 1, no. 4, pp. 74-83, Oct.-Dec. 2002

[17] Saif, U., Pham, H., Paluska, J., Waterman, J., Terman, C., Ward, S.: A Case for Goal-oriented Programming Semantics, System Support for Ubiquitous Computing Workshop at UBICOMP, 2003

[18] Yokoo, M., Durfee, E., Ishida, T., Kuwabara, K.: The Distributed Constraint Satisfaction Problem: Formalization and Algorithms, IEEE Transactions on Knowledge and Data Engineering, vol. 10, no. 5, pp. 673-685, Sept.-Oct. 1998

[19] Yokoo, M., Katsutoshi, H.: Algorithms for Distributed Constraint Satisfaction: A Review, Autonomous Agents and Multi-Agent Systems, vol. 3, no. 2, pp. 185-207, 2000

[20] Xu, D., Nahrstedt, K., Wichadakul, D.: QoS and Contention-Aware Multi-Resource Reservation, Cluster Computing, vol. 4, no. 2, pp. 95-107, 2001