Use ASAP to Develop Flexible and Efficient Distributed Self-Adaptive Systems

Ren-Song Ko and Matt W. Mutka

Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824

{korenson, mutka}@cse.msu.edu

March 28, 2003

Abstract

With their physical capability and computing power, robots can extend the capability of human beings to where it is hard to reach and handle. Robots usually need to be robust and responsive. Furthermore, given the fact that the deployment of computers increases and becomes ubiquitous, a robot may be able to accomplish a big task more efficiently through distributed computing. One approach for these challenges is to design self-adaptive systems, such as robots, that may diagnose, overcome, and adapt to various conditions without down-time.

The adaptive software architecture project, or ASAP, helps people develop and deploy adaptive distributed Java applications that may respond to different scale of environmental change. In this paper, we illustrate how to use ASAP to realize adaptive robotic systems. We also discuss the performance issues of reassembly at small temporal scale of environmental change and propose two schemes, partial reassembly and caching, to improve the performance. Our experimental results show that the cache improves the reassembly speed by a factor of $7 \sim 40$ and the time for reassembly is constant and hence predictable.

Key words: adaptive software, robot, components, constraints, Java

Relevant technical area: embedded system, programming methodologies and software

1 Introduction

It is difficult to ensure the real-time capability of embedded systems within dynamic heterogeneous environments. Under such an environment, there may be various systems with different architectures connected via networks with different bandwidth and quality of service. Thus, a priori

knowledge of available resources is not always feasible during the development stage. A robotic system is one of the examples. Analogous to computers, many roboticists claim that the use and availability of robots is going to be wide spread, in every house and business. Along with their physical capability, the embedded computation and communication capability allows robots to perform monotonous routine tasks, handle hazard materials [1] or explore distant unfriendly locations [2].

Although the tasks of robots are usually performed in real-time, the difficulty to model the real world and the environments during the system development stage may cause disastrous consequences due to the failure of the mission-critical systems. Besides, due to the research progress in teleoperation, robots may be remote controlled via networks such as Internet [3], where the environments are sensed and fed back in the form of supermedia (force, video, audio, haptic, temperature and others) for people to steer the robots. However, due to the unpredictable traffic and time delay of Internet, the interaction between robots and human being may not be responsive. The teleoperation may become inefficient and systems may even fail.

Furthermore, given the fact that the deployment of computers increases and becomes ubiquitous, a robot may be able to accomplish a big task more efficiently through distributed computing. Consider a scenario that instead of carrying all possible but may not be used equipments, a robot may only have basic function and then seek for free machines on-site to collaboratively accomplish the task. For example, a less-equipped robot may look for better sensors, more powerful computers, faster networks, and/or other robots to form an *ad hoc system* [4], which is a temporarily organized distributed system and software will be automatically distributed to each participating machines, to accomplish the task with more accuracy and better quality in less time.

One approach for these challenges is to design self-adaptive robots that control systems may diagnose, overcome, and adapt to adverse and rapidly changing conditions. Thus robots may have some adaptive features include tolerance for sensor inadequacy, environment adaptation, tolerance to actuator failure, and goal-driven choice of behaviors. Besides, telerobotic systems over low bandwidth networks, may reduce the less significant feed back or sampling rate to improve the responsiveness. An example is the "Ariel 2" autonomous legged underwater vehicle of the Air Force Research Laboratory Information Directorate and IS Robotics of Somerville, Mass., that demonstrates the feasibility of software that automatically adjusts to failures and changes in a

system [5].

Self-adaptive robotic systems are usually developed by using the behavior-based approach that does not require an accurate model of robots and environments [6]. Rather than attempting to model the world, the systems have multiple lower and more critical level control modules that react directly to sensory information and then complicated behaviors may be achieved from these low level control modules. These control modules must be pre-defined during the development stage and probably pre-loaded before execution. Therefore, the control software developers need to predict all the challenges and conditions that the robotic systems may meet. However, sometimes it is impossible to make such a prediction and the robotic systems will not be able to handle the unexpected situations without terminating the robotic systems for adding the control modules for the unexpected situations. Although the control modules may be loaded during the run-time, the availability of the control modules still need to be known for module selection algorithms to select. Since module selection algorithms are usually hard coded during the development stage, it is impossible to avoid the down-time for adding newly developed modules and updating the module selection algorithms. Furthermore, some situations are rare and the memory, which may be a scarce resource on robotic systems, for the control modules to handle those rare situations may not be well utilized. Thus, there needs to be a flexible approach for the adaptive robotic systems that will dynamically integrate new and select the appropriate control module to handle the situation they have ahead.

The adaptive software architecture project, or `ASAP` [7] [8], helps people develop and deploy distributed Java applications that may customize themselves, based on specified constraints, to multiple computing environments on different environmental change scale, i.e., how radically does the environmental change. When necessary, the execution of the applications will be temporarily suspended and check the specified constraints to search for more appropriate implementation of components, the building blocks of the applications under `ASAP`. After the searching, the application will be reassembled from the new components and resume the execution. One of novel features of `ASAP` is that `ASAP` allows newly developed modules to be added and selected without the necessity of updating the module selection algorithms and therefore the down-time. Furthermore, `ASAP` may dynamically distribute the modules or tasks to appropriate participating machines in order to

realize ad hoc systems.

For the adaptive robotic systems under ASAP, the behavior-based approach will be realized by the reassembly process if each control module is envisioned as an implementation of an abstract component and the situations need to handle are implemented as constraints. We will briefly describe the architecture of ASAP and illustrate how to use ASAP to realize the behavior-based approach in the next section. Besides, the reassembly process may have a significant performance impact on the execution of the applications, since it occurs during the execution of the applications. The reassembly process does not apply well to small temporal scale of environmental change, i.e., relatively highly frequent computing environmental changes. Section 3 describes how we improve the component reassembly performance. Section 4 describes the application, the simple robot application, running under an XR4000 robot [9], and evaluate the reassembly performance. Finally, the last two sections will give a summary, survey of related work, and then discuss potential future investigations.

2 Adaptive software architecture project

The central themes of ASAP are component, constraint, and assembly. That is,

- An application is composed of components.
- A component may have more than one implementation.
- Each implementation may have a set of constraints embedded, which distinguishes itself from other implementations and is used to determine whether its execution fits the given environment.
- Assembly will resolve, on the fly, what components an application has and what implementations each component may have.
- For each component, assembly will load each implementation and check its constraints. If all constraints are satisfied, the implementation is feasible, i.e., the execution fits the given environment.
- The application will be assembled from the feasible implementations and begin to execute.

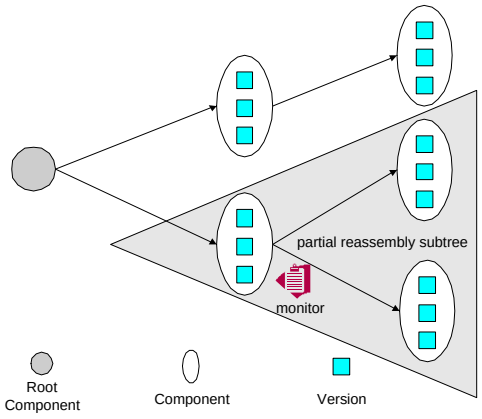

Figure 1: General ASAP software hierarchy tree

FRAME [7] of ASAP is a framework, based on Java techniques, which provides APIs for Java applications to be assembled from the feasible implementation of the components, based on whether the embedded constraints match the given environments. Therefore, applications are customized for the given environments, and hence adaptation.

2.1 Components

Under FRAME, an application should be composed of components that are implemented as Java packages and will not be assembled until execution. Each component provides services to cooperate with other components. The services define the dependency of the components and form a software hierarchy tree similar to fig. 1; that is, a parent component requires services from its child components, and vice versa. Each component, except for the root component, might have more than one implementation or version of the provided services. Only one version of each component is needed to execute a program. Each implementation of a component will produce reasonable performance or appropriate response for some specific environments. The software hierarchy information needs to be registered to a database server called the *component registry* and, of course, the component assembly needs to resolve whole software hierarchy to know which components are required for an application during run-time by querying the component registry.

As an example of FRAME, fig. 2 is the software hierarchy of a simple robot application that will retrieve obstacle information from sensors and then respond according to the type of obstacle ahead.

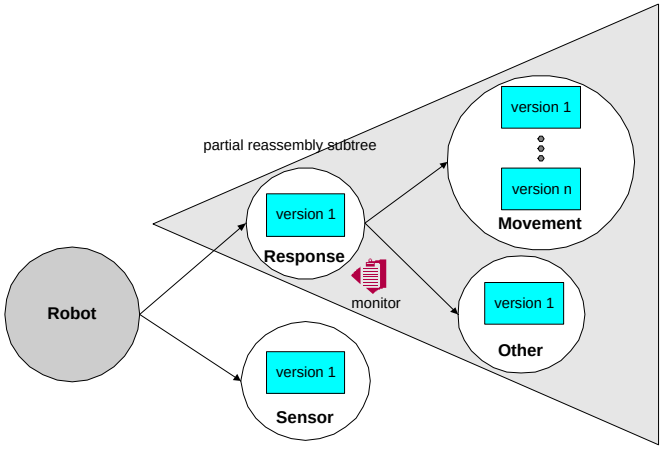

Figure 2: Software hierarchy of the simple robot application

Each low level control module is envisioned as an implementation of a component. The application consists of five components, and the component `robot` is the root component that needs service from its child components, `sensor` and `response`. The `sensor` component will collect the information about the obstacle ahead, determine the type of the obstacles, and send the information to `robot`. The `response` component will retrieve the obstacle information from `robot`. The response of the robot are actually performed by the child components of `response`, `movement` and `other`. The `other` component handles the robot actions, except how the robot will move, which is handled by the `movement` component. In this paper, we focus on the performance of the assembly, and measure how performance depends on the number of implementations. Thus, we simplify the software hierarchy by limiting each component to one implementation except `response`, which may have up to twenty versions of implementations. Each implementation of `movement` component performs some certain pre-defined movement for the obstacle, such as left circle move or right square move to pass the obstacle ahead.

Since a component may have more than one implementation, an application may be assembled from several possible combinations of components. The goal of the assembly process is to find which combination is feasible, and the answer will be found with the helps from constraints.

2.2 Constraints

Each implementation may contain a set of constraints that specify how well the implementation will perform under certain environment. A constraint is a predicate and defined as a boolean function in Java, which assembly will use, and actually is the only way, to check if the implementation is appropriate. No two implementations should have same set of constraints because, otherwise, assembly will not be able to distinguish the implementations. Take one implementation of the `movement` component, which will move the robot in a left semi-circle pattern, as an example, one of its constraints may be {"the obstacle ahead has free space at the left side."}, which specifies such a movement implementation will perform an appropriate response for the obstacles that have free space at left side. Therefore, when the robot encounters circle obstacles, the assembly process may select such a movement to pass the obstacle. Furthermore, there is a separate constraint monitoring thread, shown in fig. 1, which will determine the run-time environment changes by examining if any one of the constraints fails and invoke the reassembly process if necessary. Since the constraints are embedded into the implementation of components, the assembly process is able to bring in the newly developed implementation and examine its feasibility without down-time.

One type of constraint is *parameter*, which specifies the quantifiable or enumerable metrics, such as performance or output quality of implementation. Parameter constraints also specify the finite domains of the metrics, in which the performance is reasonable or the output quality is desired. For example, consider the parameter specified for performance. If there exists such a value within the specified finite domain that equals to the performance, the performance is said to be reasonable. The other type of constraint is *connector*, which specifies the relationship between parameters. For instance, performance usually declines when achieving better output quality. Their relationship must be specified as a connector, say $C(p, q)$ with p and q being denoted as performance and quality. Without the connector, we may specify the highest reasonable performance and intended quality within their domains. The parameter constraints are satisfied individually, but, in reality, it is impossible for the implementation to produce the highest performance and output quality at the same time and the assembly process may select a false implementation. The truth of connectors are examined with parameters specified in their domains. For example, if there exist such values p and q within their respective specified domains that $C(p, q)$ is true, $C(p, q)$ is satisfied in conjunction

with parameter constraints.

The parameters are not limited to the performance or output quality, but may be any metrics that can be enumerated or quantized. To apply the constraints to the applications that component implementations are not distinguished by performance or output quality but other concepts such as the situations need to handle, the concepts of parameters need to be extended by enumerating or quantizing the concepts. For the robot application, there may be twenty different kinds of obstacles ahead. Therefore, the constraint, {"the obstacle ahead has free space at the left side."}, of left semi-circle implementation of the `movement` component may be specified as a parameter within a finite domain as {"$14 \leq p \leq 14$"}, where p is the parameter representing the obstacle ahead and 14 represents the obstacle ahead has free space at the left side.

2.3 Assembly

The traditional way to implement a component implementation, or module, selection under some specific conditions is to use condition statements such as if-else statements. For example, there may be nested if-else statements and each is used to decide the appropriate implementation of a component. Once an implementation is selected, execution flow may go into the inner if-else statements to select the appropriate implementation of other components. However, the condition statements approach is primitive from the software engineering perspective. The condition statements contain constraint information to select the implementation and also the software hierarchy information that is required to direct the execution flow to the condition statements for selecting the implementation of the different components. While the number of components and their implementations increase, the code tends toward so called "spaghetti code" that has a complex and tangled control structure and the software will become more difficult to maintain or modify.

The most important limitation of the condition statements approach, is that condition statements are hard-coded, so the availability of all implementations need to be known during the development stage, which implies the assumption that all the computing environments may be anticipated and correctly modeled and the corresponding component implementations are implemented. It is not flexible enough to integrate newly developed implementations for the environments unknown during the development stage without rewriting and recompiling the code, and, of course,

the down-time.

FRAME uses the different approach than condition statements, called *component assembly*, to select the appropriate implementation of components. By using the information stored in the component registry, assembly may be able to identify all the component implementation of an application, and, hence, build all the possible combinations of the application. Each combination has a set of constraints, called *a software constraint*, which consists of all the constraints from involved component implementations. Since no two implementations should have same set of constraints, the mapping between combinations and software constraints are one-to-one. By solving which software constraint is feasible, i.e., all constraints in the software constraint are satisfied, the corresponding feasible combination will be found. Conceptually, assembly constructs the condition statements by querying the software hierarchy information from the component registry, and then look for the feasible combination by solving constraint satisfaction problems.

For distributed systems, such as ad hoc systems, the component assembly process will distribute components to specified platforms before constructing software constraints. There might be more than one possibility to distribute components depending on the number of components and specified platforms. We call each possibility a *distribution*. For each distribution, we may construct all possible software constraints and determine if a feasible software constraint exists. A distribution is called feasible if a feasible software constraint can be found within the distribution, and then the application is assembled from the feasible distribution with components working collaboratively via the mechanism similar to the remote function invocation. Therefore, the component assembly process for distributed applications is to find a feasible distribution from all possible distributions.

3 Assembly Performance

There are two steps in the assembly process:

1. **Software constraints sets building**: In this step, the assembly process needs to resolve the software hierarchy, i.e., to know about the components the application has and the implementations of each component. As a consequence, all possible combinations of the component implementations will be known, and then all software constraints will be built by adding the

embedded constraints, parameters and connectors, from each involved component implementations.

2. **Constraints solving**: In this step, the assembly process will find which software constraints are feasible by checking the truth of each included constraints within the specified parameter domains. `FRAME` uses a backtracking algorithm [10] for solving constraint satisfaction problems, i.e., use multiple nested loops for parameters, which take each values in the specified parameter domains and examine if the software constraint is satisfied.

The performance for software constraints sets building, denoted as P_b, depends on the number of software constraints sets and the number of constraints needs to add to each set, i.e. the summation of constraints embedded in each involved component implementation. With the backtracking algorithm, the performance for constraints solving, denoted as P_s, depends on, similar to P_b, the the number of constraints sets and the number of constraints in each set, and it also is proportional to the range of each parameter domain. That is,

$$P_b \propto \sum_{i}^{\text{all possible sets}} |S_i| \quad \text{and} \quad P_s \propto \left(\sum_{i}^{\text{all possible sets}} |S_i| \right) \times \left(\prod_{i}^{\text{all possible parameters}} |D_i| \right) \tag{1}$$

where S_i is constraints set i, D_i is the domain of parameter i.

The basic idea for extending assembly to reassembly is to use a separate thread to monitor the computing environment change and re-invoke the assembly process whenever necessary. One challenge for reassembly is performance, especially on a small temporal scale of environment change, since the assembly process involves I/O activities, such as communication between component registry, and intense computation, such as constraints solving to find the feasible combination. In our experiments, the assembly process of the robot application is about 650 times slower than the similar application hard coded by if-else condition statements. Therefore, it will be not feasible to simply re-invoke the assembly process for the reassembly. Here we propose two schemes, partial reassembly and caching, to improve performance.

First, we observe that not all components need to be changed for reassembly and it is unnecessary to examine the constraints of these components. For instance, the `sensor` component of the robot application may not need to be changed since the sensor equipment of the robot does not

change during the execution of the application. However, when the robot encounters an obstacle ahead, it needs to give an appropriate response and the `response` component may need to be changed. Therefore, as shown in fig. 1, one or more subtrees of the software architecture may be specified for partial reassembly. A monitor thread is associated with each reassembly subtree to check if all the constraints of the subtree are satisfied. Developers may also specify only the subset of all the constraints contained in the subtree to be examined to reduce the run-time performance impact by the monitors. The reassembly process will start when some of monitored constraints fail.

The other performance improvement is to use cache, which may be done in two different levels. The first level is to cache the software constraints set building, i.e., the first stage of component assembly. Since the purpose of the software constraints set building is to build all possible software constraints from all component implementations. If no component implementation is added or removed, the possible software constraints set will remain the same. By caching the software constraints set, it will be unnecessary to re-build the set if no status of component implementations change, and the first stage of the component assembly may be avoided.

The second level is to cache the computing environment, and a more aggressive scheme based on the assumption that the computing environments will repeat again. Currently, this caching scheme is key-value and implemented in a hash table as shown in fig. 3. The key will be unique for the computing environment; that is, developers have to provide a mapping that will convert a computing environment into a unique key. The value associated with the key will be the feasible components under the computing environment. To improve the probability of cache hit, the cache table may be stored in persistence storage, and then used or shared for subsequence execution of the same application.

4 Application demonstration

In this section, we use a robot, XR4000 [9], to evaluate the performance of component reassembly. As shown in fig. 2, we only specify partial reassembly for the subtree rooted at the `response` component. The only constraint to be monitored is the type of the obstacle ahead. Thus, whenever the robot encounter an obstacle, a reassembly process may be invoked depending on the type of

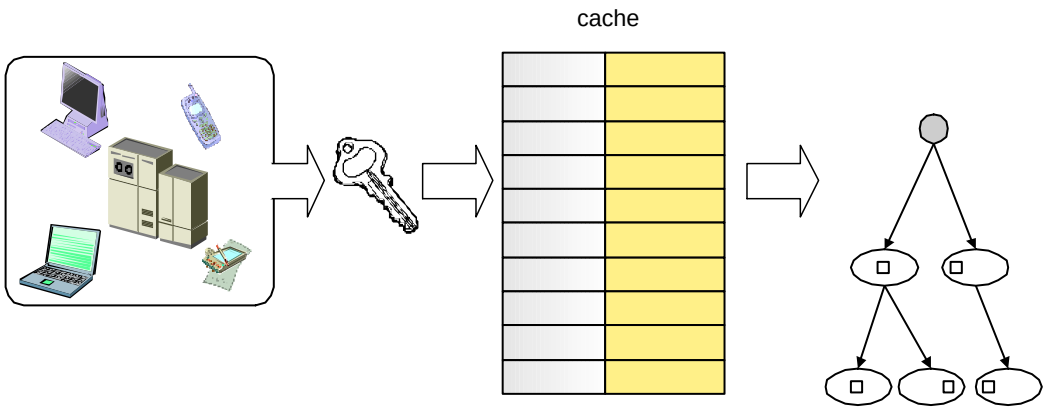

Figure 3: Flow of reassembly cache

obstacle ahead. Since only the `movement` component has multiple versions of implementations, what the reassembly does is actually to select an appropriate implementation of the `movement` component to pass the obstacle. We compare the performance of component reassembly with and without caching, and also evaluate the performance of the similar application using hard coded if-else condition statements. It needs to be noticed that, from the formulae 1, performance is application dependent, and, therefore, the performance comparison or improvement may not be same for different applications.

Fig. 4 shows that the time required for the constraint solving step, which is about $50\% \sim 60\%$ of the total time for assembly or non-cached reassembly. If the application structure does not change and no new implementation is added, the first level caching may be used, i.e., no need for software constraints sets building. The non-cached reassembly performance may be approximately reduced to the constraint solving step, which is a $40\% \sim 50\%$ time saving in this case.

Fig. 5 compares the time required to search for selecting an implementation of the `movement` component by the different scheme, i.e., non-cached reassembly, cached reassembly, and hard coded if-else statement. The if-else scheme requires about $0.003 \sim 0.018$ ms that depends on the number of implementations. The non-cached reassembly requires about $2.1 \sim 12.1$ ms that also depends on the number of implementations, and it is about 650 times slower than the if-else scheme. The non-cached reassembly time is in the order of ms and actually is good compared to the robot movement time that is usually in the order of seconds. However, for highly frequent environment changes,

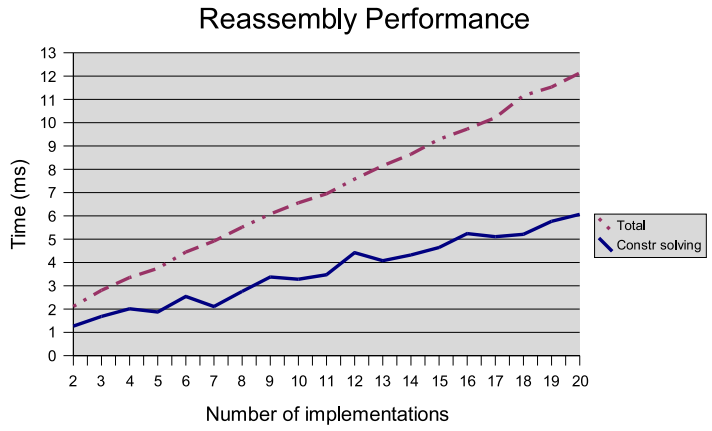

Figure 4: Constraints solving performance of reassembly

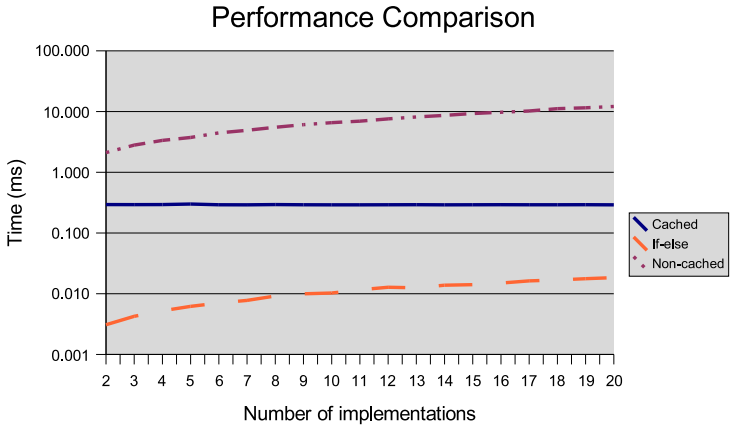

Figure 5: Performance comparison for different component selection scheme

i.e., in the order of ms, it may have significant performance impact and even may get stuck in the reassembly process since another environment change occurs before the reassembly finishes.

The result shows that cached reassembly requires about 0.29 ms independent on the number of implementations and improves the reassembly speed by a factor of $7 \sim 40$, as shown in fig. 6, and may be only about 15 times slower than if-else scheme. Unlike if-else and non-cached scheme, the cache access time is constant and independent on the number of implementations. Thus, the performance improvement becomes more significant while the number of implementations increases. Also, the constant assembly time of cache makes the execution time of the application more predictable, which is an important issue for real-time applications.

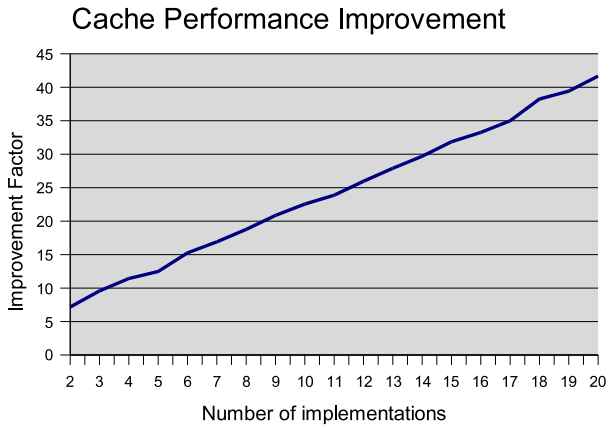

Figure 6: Performance improvement of cached reassembly over non-cached reassembly

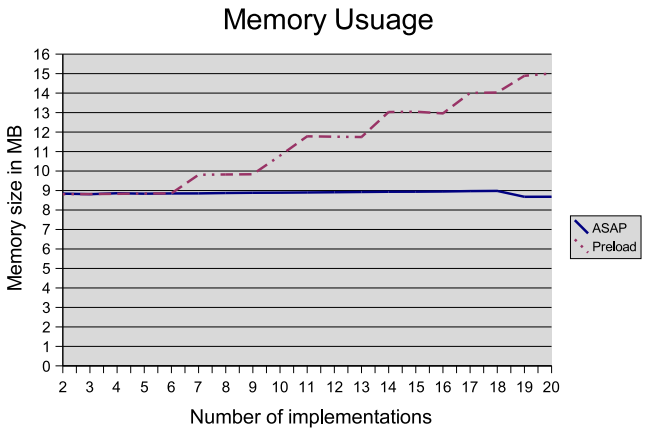

Figure 7: Memory usage comparison for `ASAP` and component-preloaded

Reassembly will load and unload the implementations of component whenever necessary, which will free some unnecessary memory, a scarce resource in embedded systems. Depending on how the application is developed, reassembly may save the memory usage. For example, the robot application using hard coded if-else statements has all implementations of the `movement` component preloaded to improvement performance. However, this is a trade-off with memory usage. Fig. 7 shows that preloaded components require about 50% more memory than `ASAP`.

5 Related work

Early robotic researchers attempted to create robot control systems that reasoned about and planned every action. The general approach was to sense the world, build a world model, plan actions with respect to goals, and then execute the plans via motor control systems. The first major development was Shakey from SRI [11]. These systems were often brittle control systems incapable of operating outside of controlled environments. Because they required an accurate world model to reason properly about what to do, environmental or sensory noise often made them unreliable. When they worked properly, they usually only operated under controlled and limited situations.

Brooks later created several robots that were designed to operating in unfriendly environments using the "subsumption architecture" [6], in which a robot constantly observes an environment and selects appropriate low-level operations depending on the environmental change.

Pham, et al. have created two systems [12] that facilitate the creation of self-adaptive control software: `PB3A` and `RAVE`. `PB3A`, the Port-Based Adaptable Agent Architecture, is a Java-based programming framework that aims to facilitate the development and deployment of self-adaptive, distributed, multi-agent applications. `RAVE`, the Real And Virtual Environment, is a mixed-reality simulation environment for mobile robots. Together, these two systems allow for the creation, testing, and analysis of self-adaptive control software by on- and off-line simulation. `PB3A` decomposes a complex systems into a hierarchy of fundamental unit of execution called Port-Based Module, `PBM`. Two `PBM`s are linked by mapping their input and output ports. Similar to `ASAP`, `PBM` may be dynamically loaded on demand across a network, and the control software can completely change its structure of the executing robot program by remapping ports or instantiating new `PBM`s. Unlike `ASAP`, the down-time and disruption is unavoidable for `PB3A` to integrate newly developed modules into running applications.

For multi-robot systems, adaptive control systems become more important because the possibility of system failure is higher than single robot system. Furthermore, the coordination communication between robots may be expensive and unstable. Yamada, et al. propose an adaptive action selection without explicit communication for multi-robot box-pushing, which changes an available behavior set depending on a situation [13], i.e., the existence of other robots and the task difficulty.

Also using the behavior-based approach, a set of behaviors or actions is designed for each situation. Each robot may determine its own situation by itself without explicit communication with other robots.

Software engineers have pursued many techniques, such as specification languages and object-oriented analysis and design, for achieving the software's original promise – applications that retain full plasticity throughout their life-cycle and that are as easy to modify in the field as they are on the drawing board. The article [14] examines the fundamental role of software architecture in self-adaptive systems and outlines several design issues, such as open or closed-adapted, type of autonomy, frequency, and cost effectiveness.

6 Conclusion and future work

We illustrate how to use `ASAP` to realize behavior-based approach to the adaptive robotic systems discussed above in the paper. The idea is that each control module is envisioned as an implementation of an abstract component and the situations need to handle are implemented as constraints. With constraints embedded in the implementations, the assembly process is able to bring in the newly developed implementation and examine its feasibility without down-time.

We have discussed the performance issues of reassembly at small temporal scale of environment change and propose two scheme, partial reassembly and caching, to improve the performance. Our experimental results show that the cache improves the reassembly speed by a factor of $7 \sim 40$ and the time for reassembly is constant and hence predictable.

One possible extension of `ASAP` is to use multiple registries, instead of one global centralized registry, which may have the location-dependent implementations that are specifically used within the neighborhood of the registry. It requires that an application is able to dynamically locate and contact an appropriate registry. Several systems perform service discovery [15] [16] [17] [18] [19], where clients generally have an a priori specification of required services. Not only in the spatial domain, it is also possible to extend the registry in the time domain, where the implementation information stored in a registry may change over time. Thus, people may design a task flow plan that will assign certain tasks to robots at a given time, and therefore may be used to coordinate multi-robot systems, in which multiple robots collaborate for complex tasks.

From the formulae 1, the constraints solving performance, P_s, depends on the number of constraints sets and the number of constraints in each set, $\sum |S_i|$, and it also is proportional to the range of each parameter domain, $\prod |D_i|$. To improve the backtracking algorithm, if more information may be extracted from the relationship between parameters, i.e. connectors, some redundancy may be found in the constraints sets. Thus, truth checking for some values in domains or some constraints may be avoided, which will reduce the terms $\prod |D_i|$ and $\sum |S_i|$ respectively.

One important aspect of ubiquitous computing is the existence of disappearing hardware [20] that are mobile, have small form factor and usually limited computation resource. Since the constraints solving may require a lot of computation, these disappearing hardware may not have enough resources. One solution is to use a dedicated server for the off-site assembly process. Therefore, the participating platforms may send the environment information to the server for assembly, and retrieve assembly result and the appropriate implementations of the components.

Finally, we may consider to implement ASAP on small devices, ranging from pagers and mobile phones to set-top boxes and car navigation systems. Sun Microsystems introduced the Java 2 Platform, Micro Edition [21], a set of specifications that pertain to Java on small devices. Some of features eliminated from Java Standard Edition include reflection, the Java Native Interface, and user-defined class loader [22]. These features are heavily used in current ASAP implementation. To port ASAP with J2ME, the entire architecture may need to be reorganized.

References

[1] S. E. Everett and R. V. Dubey, "Model-Based Variable Position Mapping for Telerobotic Assistance in a Cylindrical Environment," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 3, (Detroit), pp. 2197–2202, May 1999.

[2] P. G. Backes, K. S. Tso, and G. K. Tharp, "Mars Pathfinder Mission Internet-Based Operations Using WITS," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 1, (Leuven, Belgium), pp. 284–291, May 1998.

[3] I. Elhajj, N. Xi, W. keung Fung, Y. hui Liu, W. J. Li, T. Kaga, and T. Fukuda, "Supermedia in Internet-Based Telerobotic Operations," in *Proceedings of the 4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services* (E. S. Al-Shaer and G. Pacifici, eds.), vol. 2216 of *Lecture Notes in Computer Science*, (Chicago, IL, USA), pp. 359–372, Springer, Oct. 2001.

[4] R.-S. Ko, `ASAP` *for Developing Adaptive Software within Dynamic Heterogeneous Environments.* PhD thesis, Michigan State University, May 2003.

[5] F. Crumb, "Robot demonstrates self-adaptive software techniques." Air Force Research Laboratory Public Affairs, Jan. 2000. Information available at *http://www.af.mil/news/Jan2000/n20000107_000021.html.*

[6] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, vol. 2, pp. 14–23, Mar. 1986.

[7] R.-S. Ko and M. W. Mutka, "`FRAME` for Achieving Performance Portability within Heterogeneous Environments," in *Proceedings of the 9th IEEE Conference on Engineering Computer Based Systems (ECBS)*, (Lund University, Lund, SWEDEN), Apr. 2002.

[8] R.-S. Ko and M. W. Mutka, "Adaptive Soft Real-Time Java within Heterogeneous Environments," in *Proceedings of Tenth International Workshop on Parallel and Distributed Real-Time Systems*, (Fort Lauderdale, Florida), Apr. 2002.

[9] Nomadic Technologies, Inc., Mountain View, CA., *Nomad XRDEV Software Manual*, Mar. 1999. Information available at *http://nomadic.sourceforge.net/production/manuals/xrdev-1.0.pdf.gz.*

[10] V. Kumar, "Algorithms for Constraints Satisfaction problems: A Survey," *The AI Magazine, by the AAAI*, vol. 13, no. 1, pp. 32–44, 1992.

[11] N. J. Nilsson, "Shakey the robot," Tech. Rep. 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Apr. 1984.

[12] T. Q. Pham, K. R. Dixon, J. R. Jackson, and P. K. Khosla, "Software Systems Facilitating Self-Adaptive Control Software," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2000.

[13] S. Yamada and J. Saito, "Adaptive Action Selection without Explicit Communication for Multi-Robot Box-Pushing," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1999.

[14] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems*, vol. 14, pp. 54–62, May 1999.

[15] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley, "The Design and Implementation of an Intentional Naming System," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pp. 186–201, ACM Press, 1999.

[16] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath, *The Jini Specifications.* Addison-Wesley, 1999.

[17] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz, "An Architecture for a Secure Service Discovery Service," in *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networks*, pp. 24–35, ACM Press, 1999.

[18] E. Guttman, "Service Location Protocol: Automatic Discovery of IP Network Services," *IEEE Internet Computing*, vol. 3, pp. 71–80, July/August 1999.

[19] T. Kindberg and J. Barton, "A Web-Based Nomadic Computing System," *Computer Networks*, vol. 35, pp. 443–456, Mar. 2001.

[20] M. Weiser, "The Computer for the 21st Century," *Scientific American*, vol. 265, pp. 66–75, Sept. 1991. Reprinted in IEEE Pervasive Computing, Jan-Mar 2002, pp. 19-25.

[21] Sun Microsystems, Inc., Palo Alto, CA., *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices*, May 2000. Information available at *http://java.sun.com/products/cldc/wp/KVMwp.pdf*.

[22] S. Helal, "Pervasive Java," *IEEE Pervasive Computing*, vol. 1, pp. 82–85, January/March 2002.