# Rascal – A Resource Manager For E21

**Krzysztof Gajos** KGAJOS@AI.MIT.EDU
**Luke Weisman** LUKE@AI.MIT.EDU
MIT Artificial Intelligence Laboratory, 200 Technology sq. Rm 832, Cambridge, MA 02139, USA

## Abstract

Rascal, a system for managing resources in a smart space, is presented. Rascal contributes to improvement of the software architecture for smart spaces in two major ways: it adds a level of indirection – by referring to resources in terms of services they provide rather than devices they represent – that allows the creation of device-independent applications and it arbitrates among resource requests, thus allowing independently created applications to run together in a single environment.

## 1. Introduction

It is a goal of Oxygen to build an adaptive software infrastructure that allows software components to be assembled and configured dynamically (Oxygen, 2000). The infrastructure will allow the components to be arranged dynamically in a variety of ways to ensure that goals can be achieved. Such infrastructure works well, as long as the majority of the components exist solely as software. If it is to be used in a smart environment, such as in Oxygen's E21, where many of the software components are merely proxies for real, tangible objects, care needs to be taken to ensure that the scarce physical resources are used well.

Moreover, smart space technology is becoming increasingly more robust and accessible. But still one of the major obstacles preventing this technology from spreading is the fact that different smart spaces are equipped with very different kinds of devices. This lack of consistency makes it very difficult to write portable applications for intelligent spaces. It is thus necessary to provide an extra layer of abstraction to enable the creation of device-independent applications.

Furthermore, as the work on smart spaces progresses, the number of applications that can run independently in such environments is steadily increasing. When several applications run concurrently in an environment, a problem is created when they inevitably vie for scarce shared resources. These problems can be solved by a resource management system. A properly built resource management system will provide the necessary level of indirection thus making the writing of applications simpler. It will also provide an arbitration mechanism to ensure that scarce resources are used well.

In this paper we present Rascal–a novel resource management system for the Intelligent Room (Coen, 1998). Rascal is currently being tested in the Room.

### 1.1 What Is a Resource Manager For an Intelligent Space

What we mean by a resource manager is a system capable of performing two fundamental tasks: *resource mapping* and *arbitration* . By resource mapping (a.k.a. matchmaking) we mean the process of finding out what actual resources can be taken into consideration given a specific request. By arbitration we mean a process of making sure that, at a minimum, resources are not being used beyond their capacities. At best, arbitration ensures–via appropriate allocation of resources to requests–optimal, or nearly optimal, use of scarce resources.

## 2. Rascal

As explained before, Rascal performs two major functions: service mapping and arbitration among requests for services. Rascal is composed of three major parts: the knowledge base, the constraint satisfactions engine and the framework for interacting with other software components in the Room. The motivation for major design decisions is discussed in detail in (Gajos et al., 2001). Below we present Rascal by describing its typical uses.

*Making A Request* When an agent needs resources to provide a service, it contacts Rascal requesting all the resources it needs. If the resources are available, Rascal returns a "resource bundle" containing pointers to all of the allocated resources. If some of the requested resources could not be allocated to the request, Rascal returns an empty bundle and the agent is prevented from providing its service.

*Requesting Startup Needs* When an agent with startup needs is being brought to life, it contacts Rascal while executing its constructor, and makes a request. If the request could not be satisfied, the agent abandons startup.

*Withdrawing Or Re-allocating Previously Granted Requests* If a new request is allocated a resource that was previously allocated to a different request, two things may happen: either the resource is taken away completely from the previous requester or a different resource is given to replace the lost one. In either case, Rascal contacts the "victim" agent and notifies it of loss or change of resource.

### 2.1 The Knowledge Base

Upon startup, information about all available resources is loaded into Rascal's knowledge base (if more resources become available later on, they can be added dynamically). Rascal relies on all resources having the descriptions of their needs and capabilities separate from the actual code. Those external descriptions provide a list of services that the agent or other resource can provide. For each service provided agents may in addition specify what other resources they will need in order to provide the service. For example the `MessengerAgent` that provides a message delivery service will need one or more resources capable of providing text output service. Agents may also specify their startup needs, i.e. a list of requests that need to be fulfilled for the agent to exist.

Hence, when Rascal considers candidates for a request it not only needs to make sure that those candidates are adequate and available – it also needs to make sure that the needs of those candidates can be satisfied and that the needs of the resources satisfying the needs of the candidates can be satisfied as well, and so on. This request chaining proves to be extremely valuable: when the email alert agent, for example, requests a text output service, several different agents are considered: for example the LED sign and the speech output. The email alert agent may have its own preference as to what kind of rendition of the text output service it prefers. However if the communication link with the actual LED sign is broken, the needs of the agent controlling the LED sign will not be satisfied and so it will not be assigned to the request.

### 2.2 Cost-Benefit Analysis

When resources are scarce, part of the arbitration process is deciding which requests are more important. In Rascal self-assigned need levels are used in conjunction with the concept of utility of a service to the requester and the cost to others. This is a very simple and arbitrary scheme. It could easily be replaced by a different system should there be a need for that (again, see (Gajos et al., 2001) for motivation and (Gajos, 2000) for details).

The arbiter has to make sure that whenever it awards a resource to a new request, the cost of doing so should never exceed the utility of the awarded resources to the new requester.

### 2.3 Finding The Right Solution – The Constraint Satisfaction Engine

When the knowledge-based subsystem selects and rates all candidates for requests, a constraint satisfaction engine (CSE) is invoked to find the optimal or nearly optimal configuration that would, hopefully, fulfill the new request without breaking any of the previous assignments.

In order to find the right solution, a number of constraints and heuristics are involved of which the two most important are:
- respecting limits – there are limits on how many requests can share a service.
- preference to local solutions – as explained in (Gajos et al., 2001), it is sometimes necessary to change the assignment to a previously satisfied request. However, it is necessary to minimize such changes to the absolute minimum. By controlling the cost incurred, Rascal's CSE has been set up in such a way that changes to old requests are only made as a last resort and have to be limited in scope. That is, it should not be possible for a new request to cause changes to a large number of other assignments.

### 2.4 Rascal-Metaglue Connection

There are two major components to the Rascal-Metaglue connection mechanism: the `RascalAgent` and the `ManagedAgent`. The former makes Rascal's methods available to the rest of the Metaglue agents. The latter is an simple implementation of a Metaglue agent that all other "managed" agents inherit from. That is, all agents that want to make their services available through Rascal, or that wish to make requests through it.

## References


Coen, M. (1998). Design principles for intelligent environments. *Fifteenth National Conference on Artificial Intelligence (AAAI98)*. Madison, WI.

Gajos, K. (2000). A knowledge-based resource management system for the intelligent room. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA.

Gajos, K., Weisman, L., & Shrobe, H. (2001). Design principles for resource management systems for intelligent spaces. in submission.

Oxygen (2000). Mit project oxygen - software environment. http://oxygen.ai.mit.edu/Software.html.