

Constraint-Based Livespaces Configuration Management

Markus Stumptner

Bruce Thomas

Advanced Computing Research Centre
University of South Australia
5095 Mawson Lakes(Adelaide) SA
+61 8 8302 13965
{mst|thomas}@cs.unisa.edu.au

ABSTRACT

In this paper, we describe use of constraint-based methods for configuring ubiquitous workspaces. A declarative representation allows succinct, easily maintainable definitions of the dependencies inherent in setting up a meeting, and permits the use of general constraint reasoners for various standard tasks such as setting up meeting interfaces, switching between setting for different meetings, and saving and restoring settings. Personalisation techniques can be used for intelligently adapting the workspace to individual user needs.

Categories and Subject Descriptors

H5.3 [Group and Organization Interfaces], I.2.4 [Knowledge Representation Mechanisms and Methods]

General Terms

Design, Human Factors, Languages.

Keywords

Ubiquitous workspaces, LiveSpaces, Configuration, Constraint Satisfaction.

1. Introduction

Ubiquitous workspaces are future media-rich environments that employ new forms of operating systems and services to coordinate and manage interactions between people, multiple display surfaces, information, personal devices, and workspace applications [VER04]. LiveSpaces is ubiquitous workspaces approach that is addressing how physical spaces such as meeting rooms can be augmented with a range of display technologies, personal information appliances, speech and natural language interfaces, interaction devices and contextual sensors to provide for future interactive/intelligent workspaces [VER03]. New software infrastructure provides the basis for integrating, controlling and coordinating activities within these future workspace environments, and multiple workspaces can be integrated and synchronized at enterprise level. The Figure below shows the LiveSpaces environment that has been set up to support the Commander's Planning Group in the AuSPlanS (distributed joint headquarters planning) project.

Experience with LiveSpaces operation has shown that a significant issue is the actual management of the workspace setup: making sure that, for example, the same (or synchronized) information is shown on the different screens in the LiveSpace,

which includes the laptops. In addition, each particular meeting will involve a specific selection of devices (screens, pointing devices, lights, loudspeakers) to be activated that is geared towards its specific needs. Certain devices may be dependent on other ones, leading to a situation where making the changes required to move to the next meeting becomes a major task in itself. This leads to the notion of considering the setup problem as an issue in its own right, requiring intelligent support for the different transitions and setup choices that make up a working and meeting-specific configuration of the LiveSpace.



In this paper, we propose the use of constraint-based configuration methods for modeling a complete application environment. Configuration problems have been studied as their own active research area since the 1980s (and in fact technical configuration has been at the forefront of AI applications. The current state of the art employs constraint based representations to describe the domain knowledge [MAI98,SFH98]. Configuration is probably the most advanced AI application area in terms of using purely declarative representations for industrial purposes. Intelligent configuration researchers are working on software configuration but typically in terms of fixed compile-time building blocks at operating system level. Here we are looking at an integrated system that also administrates session-level instance information. The use of a simple unified knowledge representation scheme distinguishes us from systems like the MIT iRoom project that uses an agent-based approach with separated responsibilities rather than a unified scheme that can be used for multiple reasoning tasks.

2. Basic LiveSpace Building Blocks

Configuration is generally defined as the problem of designing a product using a set of predefined components to solve a particular task while taking into account a set of restrictions on how the components can be combined. In more complex cases,

the fixed set of components is replaced by the notion of a set of component types, typically organised in a class hierarchy together with a declarative description of how the component types relate, thereby providing an ontology of the domain.

To model a comprehensive application environment, we are dealing with five groups of entities: data, devices, applications, users, and processes (meetings).

Data simply refers to information that is passed on to applications; this includes existing files, but can also mean the set of inputs required to obtain a particular state of the application. Different categories of data would include text or more complex documents, multimedia data, plus structured data for different types of applications, e.g., HTML documents or spreadsheets.

Most complex dependencies between the data will be set up through the application description; data constraints mostly refer to storage/backup/conversion requirements.

A **device** is any digitally controllable hardware installation providing a particular service. A special case are platforms corresponding to a server or desktop/laptop machine running applications. At this point, we are concerned with providing the correct status of devices at the point where a meeting begins; we are not dealing with dynamic status changes.

Applications and services correspond to programs running in the overall workspace. The term Services is used to refer to special purpose applications in the LiveSpaces infrastructure, e.g., speech transcription, multimedia presentation, links to personal hardware, or coordination of multiple interaction modes.

Applications are the key components where explicit instantiation will play a role: having an application installed on a machine is not the same as launching it on a file (or in fact having it running on a different machine with its window locally displayed through some emulator software). Each application carries a set of commands and parameter descriptions as part of its model and a key output of the configuration system will be the scripts that actually launch the applications, composed from these command and parameter descriptions.

Users exist essentially to store access rights and preferences. Preferences are individual constraints that specify either particular applications to be used, particular visualization or application options to be chosen, and possibly to store interaction sequences in a meeting. A particular user or set of users (e.g., administrator or convener) is associated with the particular overall configuration and settings chosen for a given meeting.

5. Meeting (Process) objects will encapsulate the top level state and functionality. A meeting type can specify the types of users that would participate (e.g., developers, reviewers, managers), and the type of applications running on their respective displays or the global displays. A meeting can also specify a generalized workflow, i.e., a sequence of tasks to be fulfilled – applications run, inputs provided etc. There, saving the state of a meeting would require saving the state reached in the meeting's workflow.

3. Modeling Configuration Problems

Configuration problems have been studied as their own active research area since the early 1980s and traditionally were at the forefront of AI applications. The current state of the art employs declarative constraint based representations to describe domain knowledge for a wide range of industrial applications [MAI98,FLE98]. Configuration problems are typically solved in terms of defining the knowledge base (the set of components and

the constraints between them), the specification for a desired system (the functionality or the key components required for the system) and automatically generating a configuration (i.e., set of components and connections) that satisfies all constraints.

3.1 Components

The component types which may be used to build a configuration are described in a **component library**. Many implementations use UML class diagrams as the base language (description logics being another option). Typically, it is not known beforehand how many components are needed for a particular configuration. Instead, these components are chosen (i.e., generated) on demand during the configuration process.

Component classes are organized in an inheritance hierarchy, which can be exploited during the configuration. The leaves of the tree are the concrete component types available for the systems to be configured. Properties and behavior of a component type are defined by its attributes, ports, and constraints. There are three kinds of *component functionality*:

- functionality associated with a *certain type of component*, i.e., there has to be a key component [MF89] in the system which possesses a certain named property (e.g., the printer's ability to print, a mouse's to point, or an editor's to display HTML files).
- *value-oriented* functionality, i.e., a given component must provide a certain value for a special component attribute (e.g., a spreadsheet that computes error rates).
- *structural* functionality, i.e., the requirement to establish a connection to another component via so-called *ports* (e.g., company procedure requires that a member of each programming team subgroup be present at the review).

Each component type may have a set of **attribute** declarations. An attribute declaration consists of the name of the attribute, its type (e.g., Number, String, Enum, Boolean), and optionally a default or constant value.

Each component in a configuration has its set of individual attributes. Setting these attributes to correct values is one of the tasks in a configuration process. It may be done manually by the user or automatically by the configuration engine.

Ports are a universal concept for establishing connections between two components. Each component type may have a set of port declarations. A port declaration consists of the name of the port, its type, and its domain. Ports are themselves objects which may have attributes and are embedded in a type hierarchy. The domain of a port is a set of component types that may possibly be connected to this port.

3.2 Constraints

Constraint Satisfaction Problems (CSPs) are a powerful representation and reasoning scheme for configuration problems [SFH98]. They are a natural way to express compatibility knowledge between components. In addition, their properties facilitate the maintenance of knowledge bases:

- The same constraint can be used both for generating and checking a configuration.
- Simplicity and declarative nature of the basic constraint model allow the definition of clear semantics.
- Constraints support a clear separation of strategic and problem knowledge as well as preferences, since strategies and preferences can be expressed declaratively in terms of variable

and value orderings (e.g., seat people at the table from left to right) without changes to the constraints themselves.

- Due to their declarative nature, constraint knowledge bases can be effectively debugged using declarative methods [FEL04]

A constraint is a statement that establishes a relationship between the values of one or more attributes or ports of one or more components, e.g., *“Display the code file on the higher resolution wall projector and the video teleconference on the lower resolution one.”*

Constraints are always defined locally at one single component, from where they can reference neighbor components and therefore may navigate through the whole configuration, e.g., the user in this case: *“The code display and the code analysis window of a given user must refer to the same code file.”*

Certain specifications are of the form of performance requirements. Such “resource constraints” can be formulated by stating constraints on sets of variables, e.g., *“The number of lines of code planned for review in a has to be less or equal than the number of hours planned for the meeting times 300.”*

3.3 The Inner Workings

Formally, a CSP is defined as a set of variables with domains, with constraints defining subsets of the space of possible value assignments. Generative Constraint Satisfaction Problems (GCSP) are the theoretical foundation for the representations used in state of the art configuration tools [FLE98, MAI98]. They represent an extension of CSP’s that reifies complex objects in the context of CSPs by defining a variable set of *component variables C*. Like a normal CSP, a GCSP is *solved* when each variable is assigned a value and all constraints are satisfied. However, the set of active variables may grow during the search for a solution (and shrink again during backtracking). For a more detailed discussion of GCSPs and their semantics, see [SFH98].

GCSP’s can be solved using a backtracking constraint solver that creates components as needed during reasoning, and also observes the special semantics of property variables (which are created depending on the type assigned to a component). They can use adapted versions of the typical domain-independent propagation and filtering algorithms such as forward checking or consistency algorithms, and using the common array of heuristic control knowledge utilized for solving CSPs.

4. Operating the configuration system

A number of different service types could be gainfully provided by a livespace configurator.

Prespecified meetings: Essentially, a meeting would be specified in terms of display devices, applications, platforms, and users expected, and the appropriate set of connections and applications would be generated by the constraint engine. The base case for configuration.

To **switch** the LiveSpace to a different meeting would consist of a reconfiguration problem that would aim at leaving the maximum number of items in place. This is essentially a reconfiguration problem solvable by identifying conflicts with the current configuration and rectifying them [FEL04].

To **save a meeting and restart** the next week would require saving the current configuration; upon restart the existing constraints would have to be checked. Additional constraints may have been specified or components may have to be removed from

the configuration (e.g., a particular user cannot participate, or has to participate via a remote connection), requiring a reconfiguration process (an active area of research).

The declarative representations also permits **better health monitoring** of the LiveSpace using declarative diagnosis methods during runtime and interactive presentation of options to the user.

Until now we have assumed that the set of features chosen for a particular meeting would be either pre-specified or essentially free to choose during the meeting (different lighting or display settings to be used). A useful extension will be for the system to **learn typical modifications** made during particular meetings (or types of meetings) and making them available. This trend is mirrored by the current evolution of industrial configuration systems into personalized recommender systems.

So far we have also assumed that the requirements imposed on a particular meeting will always be satisfiable. If that is not the case, the problem changes into a Constraint Satisfaction Optimization Problem that is based on ranking among multiple imperfect solutions rather than finding a perfect one.

5. Conclusion

Ubiquitous workspaces (such as LiveSpaces) are extremely complex systems and require expert assistance to realize their full functionality. Constraint-based systems have proven up to this task in other domains. Traditionally, constraint satisfaction has been used in interface research mainly interactively for visual arrangements (e.g.,[MAL89]). We are working on using the same constraint representation mechanisms at all levels of a ubiquitous interface architecture, seamlessly integrating user modeling, hardware and software configuration, as well as generic setups and adaptations of individual meetings.

6. REFERENCES

- [MAL89] J. Maloney, A. Borning, B.Freeman-Benson: Constraint Technology fur User-Interface Construction in ThingLab II. OOPSLA 1989: 381-388
- [FEL04] A. Felfernig, G. Friedrich, D. Jannach, M. Stumptner, Diagnosis of Large Configurator Knowledge Bases", Artificial Intelligence 152(2), 2004, pp. 213-234
- [FLE98] G. Fleischanderl, G. Friedrich, A. Haselbock, H. Schreiner, M. Stumptner. Configuring large-scale systems with generative constraint satisfaction. IEEE Intelligent Systems, 13(4), Special Issue on Configuration, Juli/August 1998.
- [MAI98] A classification and constraint-based framework for configuration D. Mailharro AI EDAM, Volume 12, Issue 04, September 1998, pp 383-397
- [SFH98] Generative constraint-based configuration of large technical systems M. Stumptner G. Friedrich A. Haselbock AI EDAM, Volume 12, Issue 04, September 1998, pp 307-320
- [VER04a] Vernik M.J., Johnson S., Vernik R.J. (2004) "e-Ghosts: leaving virtual footprints in ubiquitous workspaces", Australasian User Interface Conference, Dunedin NZ.
- [VER03] Vernik, R., Blackburn, T. and Bright, D., (2003): Extending Interactive Intelligent Workspace Architectures with Enterprise Services. Proc Evolve2003, Enterprise Information Integration, Sydney, Australia, 2003.