From the Proc. of the 8th Inter. Symp. on System Configuration Management, LNCS 1439, Springer-Verlag, Berlin, 1998

System Modeling Resurrected*

André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
{andre,dennis,alw}@cs.colorado.edu

Abstract. Over the past few years, research into system modeling has dwindled in favor of other interests in the field of configuration management. Outside influence, in the form of the emergence of the discipline of software architecture, demands that renewed attention is paid to system modeling because it places new requirements on, and offers new opportunities to, system modeling. In this paper we investigate these requirements and opportunities in more detail.

1 Introduction

Version models, system construction, and system modeling have historically been the main issues for configuration management. A large amount of research has been carried out in these areas [1, 3, 5, 8, 10], of which a significant portion has made the transition into commercial configuration management systems. The focus of research has now shifted towards other interests. The introduction of software process support in configuration management systems [4, 7], the development of distributed configuration management systems [6, 14], and the creation of unified version models [2, 17] are currently among the more prominent concerns.

Despite the change in focus, research to develop new version models and provide better system construction tools certainly has continued [15, 16]. However, the lost issue in this transition seems to be *system modeling*. System modeling is the activity of describing the structure of a system in terms of its components and the relationships among them. Virtually no attention has been paid to system modeling since the development of DCDL [11] and PCL [13].

The goal of this paper is to resurrect system modeling. We advocate that it should, once again, be one of the primary research areas in configuration management. This resurrection is warranted by the recent emergence of a new research discipline, software architecture. In particular, the discipline of software

* This work was supported in part by the Air Force Material Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F30602-94-C-0253. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

architecture has been developing advanced, high-level architecture description languages that could be used by configuration management systems to advance their system modeling capabilities. These languages provide new opportunities and requirements for system modeling that were previously unrecognized, but which should now be addressed by the configuration management community. The experience base that has accumulated can be leveraged towards the development of new, integrated system models.

2 Software Architecture

System models in software architecture are often encapsulated in an architecture description language (ADL). In a typical ADL, a system is modeled as a set of primary, course grain components. These components are then combined into a complete system by explicitly modeled connections. To illustrate this mechanism, Figure 1 presents a simple client-server system that is modeled in Rapide [9], an ADL that is fairly representative of the other ADLs that have been developed to date. Shown are two components, a `Client` and a `Server` component. Each component is modeled with an interface that specifies both the functionality that is provided by the component and the functionality that the component expects to be provided by other components. In Rapide, these functionalities are specified using events. The `Server` component, for example, is capable of receiving and processing two events: `Initialize` and `Compute`. In turn, it produces one other event, `Result`. These events are used at the architectural level to connect components. In our client-server system, the `Calculate` functionality that is required by the `Client` component is attached to the functionality `Compute` that is provided by the `Server` component. This implies that whenever the `Client` component generates a `Calculate` event it is received by the `Server` component as a `Compute` event.

Most ADLs not only model components and connections, but they also explicitly model the interaction behavior of a component. In Rapide, this is done by specifying the relationship between the events that a component receives and the events that it produces. The behavior of the `Server` component, for example, is such that it produces a single event, `Result`, for each `Compute` event that it receives. Also, from the specification it should be clear that the `Initialize` event is simply consumed by the `Server` component; no events are produced by the `Server` component as a direct result of receiving an `Initialize` event.

The last aspect to be discussed about components is that they can contain constraints. These constraints specify the order in which a component can process events. The `Server` component, for example, requires that it receives a single `Initialize` event before it can process any number of `Compute` events. Such constraints can be used by analyzers to verify the consistency of a system by determining whether any of the constraints are violated.

It is now time to consider what type of contributions ADLs can make to system modeling as it has been known in configuration management. We do

```
type Server is interface
action in  Initialize();
       in  Compute(Value: Float);
       out Result(Value: Float);
constraint
   match Start -> Initialize'Call -> (Compute'Call *);
behavior
   NewValue : var Float;
begin
   (?x in Float) Compute(?x) => Result($NewValue);;
end Server;

type Client is interface
action in  Result(Value: Float);
       out Initialize();
       out Calculate(Value: Float);
behavior
   InitialValue : var Float := 0.0;
begin
   Start => Initialize;
            Calculate($InitialValue);;
end Client;

architecture ClientServer() return root is
   C : Client;
   S : Server;
connect
   (?x in Float) C.Calculate(?x) => S.Compute(?x);
   (?y in Float) S.Result(?y)    => C.Result(?y);
end ClientServer;
```

Fig. 1. Example of an Architectural System Model in Rapide.

so by investigating the applicability of the individual parts of ADLs to system modeling.

Components Traditional system models have equated components to the physical parts of a system. In PCL [13], for example, components are the result of a hierarchical breakdown of a system into its constituent subsystems. Typical usage of a configuration management system implies that a breakdown into logical components is desired, even though it is not directly supported. For example, in a compiler logical components such as "lexer" and "parser" are typically the entities that developers manipulate. These are not necessarily the physical components that are present in the system model; those correspond to such entities as "abstract syntax tree" and "file i/o".

ADLs model the logical components of a system. The inclusion in system models of architectural component modeling techniques therefore improves system modeling, because developers are able to manipulate a system from the desired, logical viewpoint.

Connectors Connectors advance the state of the art of system modeling in more than one way. First, the combination of components and connectors presents an accurate architectural view of a system. Traditional system models only modeled a hierarchical decomposition of a system without modeling the relationships among the resulting components. This omission causes many problems, such as the inadvertent removal of existing relationships or the erroneous introduction of new ones. Currently, it is hard to discover or avoid such mistakes. Since connectors model the relationships among components, they provide users with explicit information to guide them in the development process, thereby avoiding some of the mistakes that would have been made otherwise.

A second advantage of connectors is the simplification of change impact analysis. The impact of changes does not have to be deduced solely by analyzing source code, but can instead be derived from the system model using architectural dependence analysis techniques [12]. This in turn simplifies build management, as less recompilation can be achieved as well. As an example, consider a connector that is implemented as a global variable. This variable is declared in a header file that is included by many source files. Normally, all source files that depend on the header file are recompiled if the global variable is changed. If the system model instead indicates that the global variable is a connector between just two components, only the source files of those components need to be recompiled.

Behavior The inclusion of behavior has an important affect on system modeling. In the past the consistency of a system could only be verified by compiling, executing, and testing the system. Behavior modeling allows for verification that is based on the system model. As a particular version of a system is selected, it can be verified by analyzers for behavioral consistency. Of course, such verification does not guarantee correctness of the eventual system, but it does avoid a certain number of mistakes in the selection process, especially when change set technology is used or if a large version space exists. As an example, consider two change sets that can be applied to our client-server system of Figure 1. The first change set removes the `Initialize` event from the `Server` component as the component now initializes itself; the second change set removes the `Initialize` event from the `Client` component. This is modeled as two change sets since other variants of the `Server` component exist that still require to be initialized. If we only apply the first change set to the architecture, a system model analyzer can detect that the `Client` component still produces an `Initialize` event that should be consumed by the `Server` component; both change sets should be included. Such types of selection mistakes can thus be avoided even before the system is compiled and executed.

Constraints Constraints serve a similar role in system modeling as behavior; they assist in the verification of a system in much the same way. After a particular selection of a system version has been made, analyzers can verify whether any constraints are violated, and therefore guarantee correctness of a system at the system model level. For example, suppose the architecture of our client-server system incorporates two `Client` components as opposed to just one. A system model analyzer can detect that the `Initialize` event is received twice by the `Server` component because it is generated by both `Client` components. The constraint set by the `Server` component is therefore violated and the system is incorrect. These types of problems are normally only discovered when a system is tested.

3 Conclusions

The research that is presented in this paper is part of a larger effort that we are undertaking to advance the state of the art in system modeling. It is our belief that a common system model can be developed that combines the strengths of the models in both the areas of software architecture and configuration management. Moreover, we believe that it is best to take a configuration management centric view in this unification process. Because system models have been in existence in this discipline for many years, a significant experience base has been accumulated that can be leveraged. More importantly, existing configuration management systems provide an operating environment that already contains system modeling techniques to which new capabilities can be attached. Therefore, we argue that system modeling should be resurrected to once again as a primary research area in configuration management.

References

1. V. Ambriola and L. Bendix. Object-oriented Configuration Control. In *Proceedings of the Second International Workshop on Software Configuration Management*, pages 133–136. ACM SIGSOFT, 1989.
2. R. Conradi and B. Westfechtel. Towards a Uniform Version Model for Software Configuration Management. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 1–17, New York, New York, 1997. Springer-Verlag.
3. J. Estublier and R. Casallas. The Adele Configuration Manager. In W. Tichy, editor, *Configuration Management*, number 2 in Trends in Software, pages 99–134. Wiley, London, Great Britain, 1994.
4. J. Estublier, S. Dami, and M. Amiour. High Level Process Modeling for SCM Systems. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 81–97, New York, New York, 1997. Springer-Verlag.
5. S.I. Feldman. MAKE — A Program for Maintaining Computer Programs. *Software—Practice and Experience*, (9):252–265, April 1979.

6. J.J. Hunt, F. Lamers, J. Reuter, and W.F. Tichy. Distributed Configuration Management via Java and the World Wide Web. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 161–174, New York, New York, 1997. Springer-Verlag.
7. D.B. Leblang. Managing the Software Development Process with ClearGuide. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 66–80, New York, New York, 1997. Springer-Verlag.
8. D.B. Leblang, R.P. Chase, Jr., and H. Spilke. Increasing Productivity with a Parallel Configuration Manager. In *Proceedings of the International Workshop on Software Versioning and Configuration Control*, pages 144–158, 1988.
9. D.C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.
10. K. Marzullo and D. Wiebe. A Software System Modelling Facility. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM SIGSOFT, April 1984.
11. B.R. Schmerl and C.D. Marlin. Versioning and Consistency for Dynamically Composed Configurations. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 49–65, New York, New York, 1997. Springer-Verlag.
12. J.A. Stafford, D.J. Richardson, and A.L. Wolf. Chaining: A Software Architecture Dependence Analysis Technique. Technical Report CU–CS–845–97, Department of Computer Science, University of Colorado, Boulder, Colorado, September 1997.
13. E. Tryggeseth, B. Gulla, and R. Conradi. Modelling Systems with Variability using the PROTEUS Configuration Language. In *Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers*, number 1005 in Lecture Notes in Computer Science, pages 216–240, New York, New York, 1995. Springer-Verlag.
14. A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. A Generic, Peer-to-Peer Repository for Distributed Configuration Management. In *Proceedings of the 18th International Conference on Software Engineering*, pages 308–317. Association for Computer Machinery, March 1996.
15. D. Wiborg Weber. Change Sets Versus Change Packages: Comparing Implementations of Change-Based SCM. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 25–35, New York, New York, 1997. Springer-Verlag.
16. L. Wingerd and C. Seiwald. Constructing a Large Product with Jam. In *Proceedings of the Seventh International Workshop on Software Configuration Management*, number 1235 in Lecture Notes in Computer Science, pages 36–48, New York, New York, 1997. Springer-Verlag.
17. A. Zeller and G. Snelting. Unified Versioning through Feature Logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441, October 1997.