MATEC Web of Conferences **44**, 01054 (2016)
DOI: 10.1051/matecconf/20164401054
© Owned by the authors, published by EDP Sciences, 2016

Software Reuse of Mobile Systems based on Modelling

Ping Guo[1,a], Ming Liu[1]

[1]*Computing Center, Kunming University of Science and Technology, Kunming, China*

Abstract. This paper presents an architectural style based modelling approach for architectural design, analysis of mobile systems. The approach is developed based on UML-like meta models and graph transformation techniques to support sound methodological principals, formal analysis and refinement. The approach could support mobile system development.

1 Introduction

Reuse is a common practice in software development, and it is critical for both cost savings and quality assurance. When properly done, reuse of artefacts (e.g., architectural styles, design patterns, source code) often improves productivity, in turn leading to cost savings. More importantly software reuse improves the quality of software, due to the reuse of quality code components and makes it possible to leverage the collective wisdom of numerous seasoned software designers and software engineers who faced similar challenges and found optimal solutions widely accepted by the software developer community.

Types of reusable software artefacts vary ranging from those found in the requirements phase to design phase to implementation and testing phases. They include domain requirements, architectural styles and design patterns, Commercial Off-The-Shelves (COTS), code reuse, reusable tests, etc.

A number of architectural styles [1] for distributed systems have been identified, such as ”client-server system”, ”message-passing protocols ”, ”event-based systems” etc. Architectural styles are often developed systematically to solve difficult architectural problems on the conceptual level. When designing a new software, the engineers can select and reuse an appropriate style(s) as the basis for further design and improvement.

With the fast development of wireless communication technologies, and the increasing popularity of portable computing devices, many [4, 5, 6] platforms and paradigms for mobile systems have been created to deal with the new requirements brought by mobility and provide better service to applications. At the same time, the design and development of mobile systems is a difficult task. Problems arise from the complexity of the mobility and the lack of abstractions, services, and tools for designing, analyzing, implementing, and testing it. However, the previously mentioned architectural styles and approaches are designed for distributed systems, and they are very hard to be used for mobile systems, since mobility has created additional complexity for computation and coordination. The current architectural approach [12] offers only a logical view of change; it does not take into account the properties of the ”physical” distribution topology of locations and communications. It relies on the assumption that the computation performed by individual components is irrelative to location of the component, and the coordination mechanisms through connectors can be always transmitted successfully by the underlying communication network. In order to support mobility, the architectural approach needs to be adjusted in different abstract layers of modelling and specification languages.

We propose to use styles for mobile systems to capture the properties of a certain class of mobile computing middleware platforms. Such a style should provide high-level abstractions for the structure, behaviour, and key properties of the platforms.

We present a methodology for developing complicated software systems based on architectural styles in this paper. The approach is developed based on graph transformation systems to support sound methodological principals, formal analysis and refinement. We illustrate the approach through the modelling and simulation of architectural styles for nomadic networks.

The rest of the paper is organized as follows. Sect. 2 introduces the framework of our approach, which includes two parts: modelling and simulation. Sect. 3 illustrates the approach through the modelling of

[a] Corresponding author: guoping29@yeah.net

architectural styles for mobile computing middleware. Related work is discussed in Sect. 4.

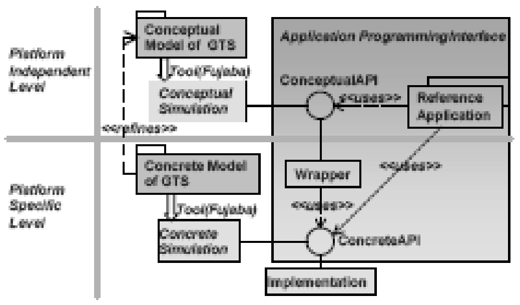

Fig.1. The modelling framework based on GTS

2 The modelling based on GTS

Graphs are often used as abstract representation of diagrams, e.g., UML class diagrams. Graph transformation is defined by the application of graph transformation rules that model the permitted actions on graphs representing system states. Such transformation defines a relation on state graphs that can be iterated arbitrarily yielding the transformation process. In this way, a graph grammar consisting of a start graph and a set of rules gets an operational semantics.

A GTS (Graph Transformation System) [13] is a system using the techniques of graph transformation. Providing visual representation and formal semantics, GTSs are exploited for modelling distributed systems, mobile systems and other complex software systems. Especially, a TGTS (Typed Graph Transformation System) is used for modelling functional requirements, software architectures, and architectural styles. We will use the TGTS [14] G = (TG,C,R) to define architectural styles. TG is here meta models (or, UML class diagrams) that define architectural elements. C is a set of constraints restricting their possible compositions. Simple constraints, already included in the class diagrams, are cardinalities that restrict the multiplicity of links between the elements. More complex restrictions can be defined by OCL (Object Constraint Language) constraints. R is a set of graph transformation rules (given by pairs of object diagrams).

In order to help the design and development of complicated software system, we develop an approach based on the GTS modelling and simulation (in Fig. 1). The approach is driven by a stepwise refinement between GTS-based models on different abstract levels. As shown in Fig. 1, a conceptual model is smaller and easier to understand; a concrete model reflects more design or implementation concerns. Starting from an abstract description of the systems, stepwise refinements yield more and more concrete specifications that should finally be directly implementable.

The definition of the model as platform-independent or platform-specific is rather relative. The conceptual model in Fig. 1 is platform-independent compared to the concrete model. The concrete model is platform-specific since it has a more specific design and realization. On the other hand, the conceptual model is also platform independent in the sense that it is independent of specific programming languages, hardware platforms and concrete implementation methods. Besides this, the number of the abstract levels is not fixed as only two levels. More or less levels can be defined according to the complexity of the system and requirements of the models.

For example, we define three-level abstract models in Sect. 3 in order to decrease the complexity of the styles. The operational semantics of the GTS allows us to execute the models and thus analyzing the system through simulation and model checking. Due to the well known state explosion problem with model checking, we prefer using simulation for analysis. Simulating the dynamic behaviour of a high level architecture can implement the system in a quicker, cheaper, and more flexible way. Some desired information and properties can be gotten through the simulation of the architecture. The programmers can execute the system and have a direct feeling how the system works.

3 Architectural styles for mobile systems

In this section, we will illustrate the proposed approach through an example of how to specify architectural styles for mobile systems. The main aspects of the mobile systems are explained in Sect. 3.1. Sect. 3.2. illustrates how to separate the aspects into different abstract levels of styles. Afterwards, we explain an example: styles of mobile systems for nomadic network to show how to get a complete specification of a style by starting from a very abstract conceptual style (in Sect. 3.3).

3.1. Main characteristics of mobile systems

When building architectural styles for mobile systems, we need to identify the main properties to be modelled. Two important aspects: dynamic change and component interaction are explained here.

3.1.1. Dynamic change Supporting for runtime modification is a key aspect of mobile systems. Mobility represents a total meltdown of the stability assumptions (e.g. network structure, network connection, power supply, CPU, etc.) associated with traditional distributed computing. The main differences are caused by the possibility of roaming and wireless connection. Roaming implies that, since devices can move to different locations, their context (network access, services, permissions, etc.) may change, and that mobile hosts are resource limited, for example, in computation power, memory capacity, and electrical power. Wireless connections are generally less reliable, more expensive, and provide smaller bandwidth, and they come in a variety of different technologies and protocols. This results in a very dynamic software architecture, where configurations and interactions have to be adapted to the changing context and relative location of components.

3.1.2. Component interaction

One important purpose of a mobile system is facilitating component interaction, and enabling the cooperation of distributed components. More specifically, component interaction covers component communication, collaboration, and coordination. These different aspects are not orthogonal and there is no clear boundary. For example, components need a method to communicate with each other through the network. The method can be message based, transactional based, event based, etc. The communication requires the coordination and synchronization between different actions and components, and the components collaborate with each other in order to perform a task.

A mobile system can be distinguished through the supported component interaction patterns and paradigms, e.g., RPC (Remote Procedure Call) pattern, message-based pattern, event-based pattern, etc. The static assumption of network structure, network connection and execution context in distributed systems makes the component interaction quite static. The components involved for interactions are generally fixed. Such fixed configurations will not change till a dynamic change happens. And the messages for communication are always supposed to be successfully transmitted by the network. In a mobile setting, the components involved in an interaction change dynamically due to their migration or connectivity patterns. This results in dynamic changes and reconfigurations among the components interaction.

3.2 The modelling framework for the styles

In order to define the style clearly and decrease the complexity brought by mobility, we will separate the styles into three abstract levels, i.e, the conceptual style and concrete styles in Fig. 2. The identified two properties: dynamic change and component interaction will be modelled on different abstract levels. The conceptual style will specify only the dynamic change of the components in the presence of mobility, but not the component interaction, for the reason that dynamic change is more general and abstract, while component interaction is more design specific. Different mobile systems can support the same model of dynamic change, whereas the applied interaction models can be very different.

For example, the conceptual style of mobile systems for nomadic networks (in Fig. 2) can be refined into concrete styles that support different component interaction models, such as RPC and publish-subscribe model. Besides this, the conceptual model should answer the following questions: What functionalities can be provided by such mobile systems? What application scenarios can be implemented on the system? On the concrete level, the conceptual style is refined into a concrete style, where more and more design-specific aspects like component interaction protocols and patterns are integrated into the core functionality. The concrete style also specifies dynamic changes of components involved in an interaction due to their migration or connectivity patterns.

As a result, the dynamic change specified on the conceptual level will be associated with component interaction on the concrete level. The concrete style can be refined into a platform-specific style to add more implementation specific aspects. For example, the platform-independent concrete style based on RPC (in Fig. 2) can be refined into a platform-specific concrete style of Wireless CORBA, while the concrete style based on Publish-Subscribe can be refined into a concrete style of REBECA. We will explain the three-layer modeling approach through specifying the architectural styles of mobile computing middleware for nomadic networks. The corresponding three abstract layers will be illustrated in the following three sections.

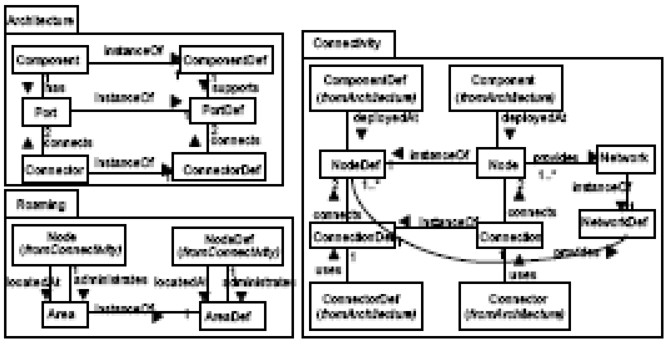

Fig.2. Conceptual style

3.3 Conceptual style

The conceptual style should define dynamic changes and main functionalities of a class of mobile computing middleware for nomadic networks. Nomadic networks utilize the infrastructure as access points to establish connectivity among mobile application components and to coordinate their transmissions. The nomadic network has a structured space where components are allowed to move inside the scope of areas covered by access points. Examples for such network are WLAN and telecommunication networks as GSM, GPRS, UTMS. The mobile computing middleware for nomadic network focuses on how to provide continuous connectivity and other services when components move across the structured spaces where handover protocols are often used.

Recalling the definition of the TGTS, G = (TG,C,R). The meta model (TG) of the style is shown in Fig. 2. We use a meta model structured into three packages. This allows us to separate different concerns, like software architecture, distribution, and roaming, while at the same time retaining an integrated representation.

Our meta models capture these relations in the three packages Architecture, Connectivity and Roaming to present different viewpoints of the systems [7]. The structured space of nomadic network is modeled as AreaDef in Roaming package. An area is defined by an administrative domain, like a cell managed by a GSM base station, or aWireless LAN domain. The wireless connection is modeled as Connection in Connectivity package. Connection is a physical network link which delivers communication services to Connectors at the software level. The typing ConnectionDef means that we can distinguish, for example, between Ethernet, WLAN, or GSM-based connections.

The Architecture package defines the architectural view, containing both a definition of an architectural model (meta classes ComponentDef, ConnectorDef, PortDef) and an individual configuration (meta classes Component, Connector, and Port), related by the meta association instanceOf. It defines the architectural elements and constraints, and it represents the TG(type graph) in a TGTS. The style includes architectural model which is defined through meta classes, e.g., ComponentDef, ConnectorDef, PortDef) as well as of an individual configuration (meta classes Component, Connector, and Port), related by the meta association instanceOf.

The conceptual style is focussed on the roaming and connectivity of mobile hosts, i.e., hosts can change location and possible connections may vary according to their relative location to each other. Naturally, architectural components and behaviour of applications depend on the connectivity and location of their host computers. A node is a (real or virtual) machine, accessible through bridges via connections. The typing means that we can distinguish, for example, between Ethernet, WLAN, or GSM-based connections, or between different kinds of machines like PCs, laptops, cell phones, etc.

A Connection is a physical network link which delivers communication services to Connectors at the software level. An area is defined by an administrative domain, like a cell managed by a GSM base station, or a Wireless LAN domain. Thus, there are different types of areas which may be overlapping. An area can have an administrative Node who serves the connectivity in this Area. This node does not need to be located inside the same area. We do not separate mobile from stationary hosts at this level. A node is mobile if it changes its location to a different area, a fact that is part of the dynamics of the model. This allows the added flexibility of considering, e.g., a laptop that does not move as a stationary device.

The Connectivity package presents the distributed view of the system in terms of the concepts Node and Connection, paired as above with corresponding type-level concepts. a (real or virtual) machine. distinguish, for example, between Ethernet, WLAN, or GSM-based connections, or between different kinds of machines like PCs, laptops, cell phones, etc. The package uses the elements Component, ComponentDef from the Architecture package to be able to specify deployment using the deployedAt association. A Connection is a physical network link which delivers communication services to Connectors at the software level.

The Roaming package defines the location and mobility of Nodes in terms of Areas, i.e., places where Nodes can be located. area is defined by an administrative domain, like a cell managed by a GSM base station, or aWireless LAN domain. Thus, there are different types of areas which may be overlapping. An area can have an administrative Node who serves the connections in this Area. This node does not need to be located inside the same area. We do not separate mobile from stationary hosts at this level. A node is mobile if it changes its location to a different area, a fact that is part of the dynamics of the model. This allows the flexibility of considering, e.g., a laptop that does not move as a stationary device.

We define graph transformation rules (R): moveIn, moveOut, connect, disconnect and handOver. These rules are used to describe the dynamic changes and reconfigurations of the style in the presence of mobility. moveIn, moveOut model the movement of components, connect, disconnect model the connection and disconnection of wireless networks, and handOver models the main functionality handOver.

As examples, we will introduce the rule handOver here, where the pre-conditions of the rule describe the situation must be satisfied when reconfiguration happens, and the post-conditions describe the results after the reconfiguration. The rule handOver (in Fig. 3) explains how to maintain connectivity between administrative domains. It requires that node n1 is located in two areas served by two administrative nodes n2 and n3. Connector cr uses connection c1 between node n1 and n2. The connection is replaced by c2 of type ConnectionDef which, according to the types declaration, is permitted between nodes of type NodeDef nd1 and nd2. The uses relation of the connector is transferred to the new connection.

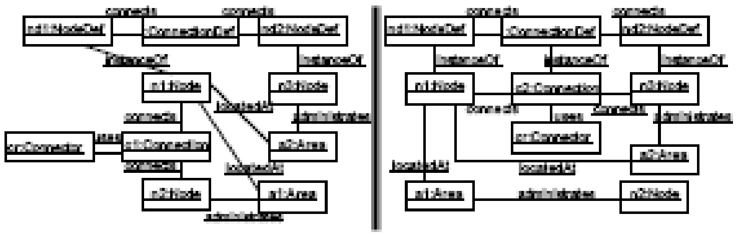

Fig. 3. HandOver rule

4 Related work

A number of different process calculi have been proposed to describe the interaction and movement of mobile processes, mostly by extending the calculus [3, 6, 8]. In order to express runtime properties of mobile systems, some of these calculi have been complemented by logics [10, 11]. Apart from the fundamentally different appearance and style which, in our opinion, makes them harder to grasp for the average software engineer than our meta model-based approach, these process calculi with their associated logics have a complementary focus: they provide means for programming mobility in terms of the processes driving individual components or devices, rather than for its high-level conceptual modelling in terms of global pre- and post-conditions. In this sense, our model defines requirements, e.g., for handOver, while the actual protocol implementing the operation is more easily specified (and verified) in a process calculus.

There exist different specification and modelling methods of mobility in the software architecture area. Some of the techniques proposed by the AGILE project [2] are closer to our approach. AGILE develops an architectural approach by extending existing specification languages and methods to support mobility: UML stereotypes are used to extend UML class, sequence and activity diagrams in order to describe how mobile objects can migrate from one host to another, and how they can be hosts to other mobile objects. Graph transformation systems are proposed as a means to give operational semantics to these extensions.

Other extensions are based on architectural description languages, like the parallel program design language CommUnity using graph transformation to describe the dynamic reconfiguration; Klaim is a programming language with coordination mechanisms for mobile components, services and resources; The specification language CASL as a means for providing architectural specification and verification mechanisms. While Klaim and CASL are more programming and verification-oriented, the approaches based on UML and CommUnity are at a level of abstraction similar to ours, but the goals are different: our focus is to model a style of mobile systems, while the focus in the cited approaches is on the modeling of applications within a style that is more or less determined by the formalisms.

References

1. G. D. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. In ACM Transactions on SoftWare Engineering and Methodology, volume 4(4), pages 319–364, Oct. 1995.
2. L. Andrade, P. Baldan, and H. Baumeister. AGILE: Software architecture for mobility. In Recent Trends in Algebraic Develeopment, 16th Intl. Workshop (WADT 2002), volume 2755 of LNCS, Frauenchiemsee, 2003. Springer-Verlag.
3. L. Caires and L. Cardelli. A spatial logic for concurrency (part I). Information and Computation, 186(2):194–235, November 2003.
4. L. Capra, C. Mascolo, andW. Emmerich. Middleware for mobile computing. In Q. Mahmoud, editor, Middleware for Communications. John Wiley, 2002.
5. L. Cardelli and A. Gordon. Anytime, anywhere. model logics for mobile ambients. In 27th ACM Symposium on Principles of Programming Languages, pages 365–377. ACM, 2000.
6. L. Cardelli and A.D. Gordon. Mobile ambients. In Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98. Springer-Verlag, Berlin Germany, 1998.
7. R. Heckel and P. Guo. Conceptual modeling of styles for mobile systems: A layered approach based on graph transformation. In IFIP TC8 Working Conference on Mobile Information Systems(MOBIS) and IFIP International Federation for Information Processing, pages 65–78. Kluwer and Springer Verlag, Sept. 2004.
8. M. Hennessy and J. Riely. A typed language for distributed mobile processes. In Proc. ACM Principles of Prog. Lang. ACM, 1998.
9. H. J. Koehler, U. Nickel, J. Niere, and A. Z undorf. Integrating uml diagrams for production control systems. In 22nd International Conference on Software Engineering (ICSE), pages 241–251. ACM Press, 2000.
10. S. Merz, M.Wirsing, and J. Zappe. A spatio-temporal logic for the specification and refinement of mobile systems. In Mauro Pezz`e, editor, Proc. Fundamental Approaches to Software Engineering, 6th International Conference (FASE 2003), volume 2621 of LNCS, pages 87–101. Springer-Verlag, 2003.
11. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. Information and Computation, 100:1–77, 1992.
12. G.-C. Roman, G. P. Picco, and A. L. Murphy. Software engineering for mobility: A roadmap. In A. Finkelstein, editor, Proc. ICSE 2000: The Future of Software Engineering, pages 241–258. ACM Press, 2000.
13. G. Rozenberg, editor. Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1:Foundations. World Scientific, 1997.