

BEHAVIOURAL MODEL OF COMPONENT-BASED GRID ENVIRONMENTS

Alessandro Basso, Alexander Bolotov, Vladimir Getov

Harrow School of Computer Science

University of Westminster

Watford Road, Northwick Park

Harrow HA1 3TP, London, U.K.

[bassoa.bolotoa,v.s.getov]@wmin.ac.uk

Abstract In component-based Grid environments, we analyse the problem of formal specification of their behaviour by introducing an automata-based model. We show how to construct this new framework from the analysis of states of components and how to apply it to a reconfiguration scenario in a dynamic distributed system environment. We aim at building a framework for future integration of these developments in a software tool for runtime automated specification and verification, ensuring a reliable dynamically reconfigurable component model.

Keywords: GCM, Grid IDE, Reconfiguration, Temporal Deontic Specification, Dynamic Verification.

1. Introduction

Component models enable modular design of software applications that can be easily reused and combined, ensuring greater reliability. This is important in distributed systems where asynchronous components must be taken into consideration, especially when there is need for reliable dynamic reconfiguration. In these models, components interact together by being bound through interfaces, however, there is a further need for a method which ensures correct composition and behaviour of components and their interaction with the environment.

Fractal [9] is a modular and extensible component model. The Fractal specification defines the Life Cycle controller interface as [8]: “A component interface to control the lifecycle of the component to which it belongs. The lifecycle of a component is supposed to be an automaton, whose states represent execution states of the component. This interface corresponds to an automaton with two states called **STARTED** and **STOPPED**, where all the four possible transitions are allowed. It is however possible to define completely different lifecycle controller Java interfaces to use completely different automata, or to define sub interfaces of this interface to define automata based on this one, but with more states and more transitions. A great number of component models in fact consider by default a number of substates to the most generic **STARTED** state, allowing for a deeper introspection on the behaviour of states of components (initialized, suspended, failed. . .).

The Grid Component model (GCM) [13] is an extension of Fractal built to accommodate requirements in distributed systems, in particular, those developed within and following the CoreGRID [12] project. The GCM specification defines a set of notions characterising this model, an API (Application Program Interface), and an ADL (Architecture Description Language) [4]. In Fractal, when changing the bindings of a component, this component must be stopped (in other words, to avoid disruption to the system, when unplugging a component, such component must be stopped before severing its connections to other components); at the same time, invocation on controller interfaces must be enabled when a component is stopped (in order to send the stop signal to the component), making de facto impossible to reconfigure the component controller. In GCM section 8.1 of [13], the life-cycle controller is extended allowing to separate partially the life-cycle states of the controller and of the content. When a component is functionally stopped (which corresponds to the stopped state of the Fractal specification), invocation on controller interfaces are enabled and the content of the component can be reconfigured. When a component is stopped, only the controllers necessary for configuration are still active (mainly binding, content, and lifecycle controllers), and the other components in the membrane can be reconfigured. We can make use of this extended capabilities and monitor the changes in states of components.

The recent development of a Grid Integrated Development Environment (GIDE) based on the GCM specification [3] opens new possibilities for the dynamic reconfiguration scenario in large distributed systems. We are able to take advantage of pre-built components in the GIDE (namely the component's hierarchical composition, their API, and the monitoring of both components and resources) to form a basis for a reconfiguration framework which exploits the underlying properties of the specification language and deductive reasoning verification methods used in our research. We consider the monitoring specification of [10] and the state information that can be retrieved through calls to the `LifeCycleController` interface (`getFcState` operation) for components, as well as other monitoring techniques for the environment.

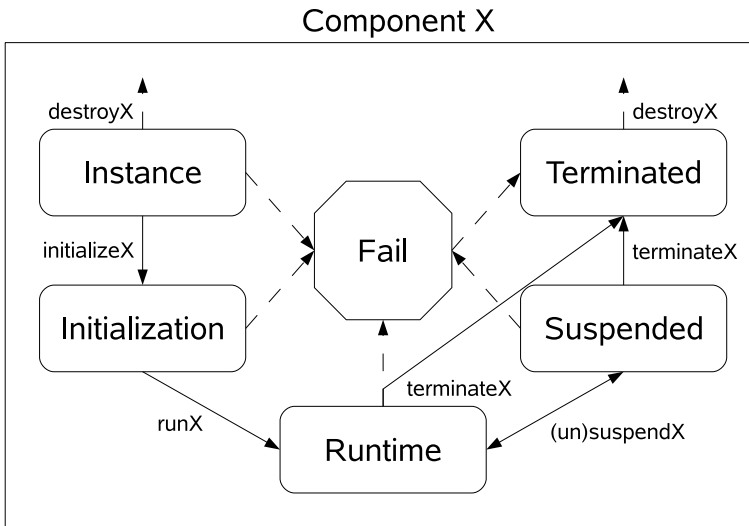
The rest of this paper is organised as follows. In §2 we give some basic information on behaviour of stateful components/resources (§2.1), analyze the limitations of ADLs (§2.2) and the environment monitoring techniques which we utilize (§2.3). Further, in §3 we introduce the automaton model used for formal specification, detailing the component level automata in (§3.1) and the environment level one in (§3.2). In (§4.1) we outline the Specification process and its usage for reconfiguration purposes is introduced in (§4.2). Finally, we give some concluding remarks and identify future work in §5.

2. Background

2.1 Behaviour of stateful components/resources

The basic lifecycle of components, and thus the resources being managed, can be retrieved at runtime by the use of the Component Monitoring and Resources Monitoring systems, built in the GIDE, through: components state calls (implemented by all component objects), specialised parameters monitoring for some specific components, resources availability monitors and others. This state system is often restricted, in that it supports the deployment processes used by the framework and models only the deployment state of the system, not its operational characteristics. Each deployment component independently represents the state of the deployed resource which it is managing. The system as a whole must also represent a reasonable depiction of the overall state of many components. The core lifecycle is defined by the states, allowed transitions and operations shown in Figure 1.

As a component is such that it conforms to a set of defined states, and to the GCM, we can therefore consider composite components as components that inherit the same properties and conform to state composition. The analysis of the components' instances becomes now crucial. When a component is in the **instance** state, this component (and all its requirements) will be deployed to the appropriate system, and any operations will be performed that are part of the components instantiation process. This state also presumes that whatever

Figure 1: Component's Lifecycle States

activation is required in order for the resource handler of the component to be valid has been performed (we will leave the detailed requirements for the resource monitoring system after more research into the effect of distributed properties at resource level). As shown in the diagram, the *initialize* and *destroy* state change commands are supported in this state. The component will then move to **initialization**, where it will wait until a call is made to run; passing on to the **runtime** state, which indicates that the services provided by the resources that are being deployed are available for use. This state does not indicate any information regarding the operational capabilities of the service, only that it has completed initialization and not failed. At any time, state actions may not complete correctly or the service itself may fail. In response to these failures, the component will transition to the **fail** state. The component may remain active in the system, but its managed resource is presumed to no longer be operational. Once the component is running or has failed it should either eventually or immediately terminated to stop its services. The **terminated** state represents a state where a component is no longer running and cannot be returned to the running state without redeployment. This state, however, does not eliminate the resource from the system. Upon invocation of the *destroy* command, the component's corresponding resource will be freed. In a system with multiple components, the lifecycle of the whole system is defined by the relationships between the individual component lifecycles. The state of each component is bound to the state of the components it relies on. The hierarchy of the

system defines relationships where related components lifecycles are linked. The component model and the ADL specification help define explicit semantics for guiding lifecycle transitions.

2.1.1 Suspended state. Further analysis should be considered into the **runtime** state above. We consider a special state in which the components may be transitioning to and from the running state. In this particular state, called the **suspended** state, special attention has to be made to the states of the resources relative to the component in question (ie. the resources may be released while a component is a suspended state). These properties help refine the way components and relative resources are handled in respect to their stateful behaviour.

2.1.2 Wait state. The case of the **wait** state is a very particular one. This particular case is often referred to when a component is ready to receive the input required for continuing its process (although some other special cases could arise depending on the specific component). This state is often fallen back into the more generic **runtime** state, since resources are not released by the component although they may not even be “used” (ex. the component may be deployed on a node but not utilizing the processing power). We are currently forced to consider this state as a particular case of **runtime** state as there are no implemented ways to monitor this situation through the lifecycle controller.

2.2 The ADL limitations

It is well known that Architecture Description Languages (ADLs) generally cannot provide sufficient insight into the post-deployment / runtime reconfiguration [14]; although they can be used to describe components, connectors and configurations as well as the hierarchical structure of the system. We have to therefore rely on specific characteristics about the states of instantiated components (also known as “live components”) using standard runtime monitoring tools. We can retrieve the specific state information (described in the previous section) as messages passed to the system thus describing the runtime behaviour of states of the component. Similarly, the overall view of behaviour of states of the components’ system and resources, describes the runtime behaviour of the environment. We use “finite state on finite strings automata” for the former and “infinite state automata” for the latter, for our runtime behaviour specification.

2.3 Environment Monitoring

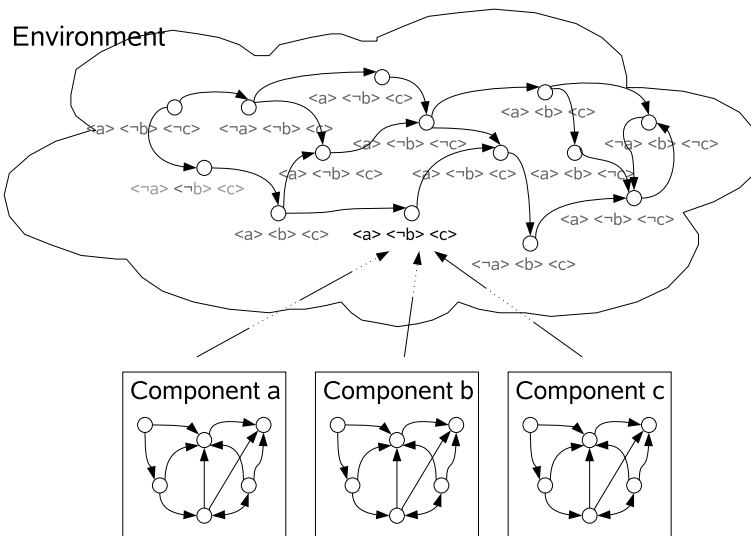
When considering the state of components and resources in a GCM model, and the runtime monitoring of the environment, we analyse the following introspections.

- For components, by accessing the `LifeCycleController` interface we are able to know the state of the requested component (namely **Started** and **Stopped**).
- For resources, we can monitor their availability status as long as these resources are specified during composition by some deployment descriptor, or at runtime some metadata provider. As the former is mandatory when using some specific components [1], it is not mandatory for all. We will assume that if the developer is interested in using this formal specification for safe reconfiguration of components, he will provide some accessibility to metadata information on runtime availability (as well as list of required resources for the corresponding components), which can be monitored at runtime.

3. Automata Based Model

In building our specification protocol, we follow well known automata constructions. We take a simple finite state automaton on finite strings, for the components specification, and a more complex infinite state on finite strings automata to define the environment. The automata at component level are used for the creation of labels defining the various states in which the considered component is, and are then fed upon request on to the various states of the automata at environment level (Figure 2).

Figure 2: Automata Based Model



3.1 Component level automata

For a component level automaton we suggest to use a finite automaton on finite words. Let Σ be a finite alphabet. A finite word over Σ is an element of Σ^* .

DEFINITION 1 (FINITE WORD AUTOMATON) *A finite word automaton, A , is a tuple $A = (\Sigma, Q, Q_i, Q_f, \Delta)$ where Σ is a finite alphabet, Q is a set of states, $Q_i \subseteq Q$ is a set of initial states, $Q_f \subseteq Q$ is a set of accepting states, and $\Delta : Q \times \Sigma \longrightarrow 2^S$ is a transition function.*

A run, R , of A over a word $w = a_1, a_2, \dots, a_n - 1$, $w \in \Sigma^*$ is abbreviated as R_w and it is a sequence of states s_1, s_2, \dots, s_n such that for any i , $(0 \leq i < n)$, $s_{i+1} \in \Delta(s_i, a_i)$. A run, $R = s_1, s_2, \dots, s_n$, is successful if $s_1 \in Q_i$ and $s_n \in Q_f$. We say that an automaton A accepts a word w if it has a successful run R_w . In this case we also say that an automaton A is not empty.

When we construct such an automaton at the component level, we would call it A_c and we assume the following:

- Initial states, Q_i , are either 'running / waiting' or none of the previous;
- The set of states, Q , corresponds to the states of the component as defined in the previous section;
- The acceptance condition is defined as reaching of one of the following states: terminated, suspended state or fail. These states are in the set Q_f and the acceptance condition is to reach one of these states in Q_f
- The transition conditions are determined by the state change calls of the component.

When the assumed automaton A_c (non-)emptiness procedure establishes that the automaton is not empty, it returns a successful run of A_c . Thus, for any component cycle, when the corresponding automaton has an accepting run, it means that a component is in the one of the accepting states. We would define a simple function $Lab(A_c)$ which returns the following parameters:

- $\langle a_t \rangle$ - when a component has met the acceptance condition "terminate"
- $\langle a_s \rangle$ - when a component a has met the acceptance condition "suspended"
- $\langle a_f \rangle$ - when a component has met the acceptance condition "terminate after going through fail state"
- $\langle \neg a \rangle$ - when component a has not met any acceptance condition

These parameters generated by the function $Lab(A_c)$ will be subsequently passed to the environmental level automata which is described in the next section.

3.2 Environment level automata

For the environment level we consider automata on infinite trees. Namely, we consider Buchi tree automata [16] which is an extension of the standard tree automaton accepting infinite trees.

DEFINITION 2 (INFINITE TREE AUTOMATON) *A Buchi Tree automaton $A_T = \langle \Sigma, D, S, M, s_0, Q_f \rangle$, where Σ is an alphabet, $D \subset \mathbb{N}$ is a finite set of branching degrees, $M : S \times \Sigma \times D \longrightarrow 2^{S^*}$ is a transition function satisfying $M(s, \sigma, d) \in S^d$, for every $s \in S, \sigma \in \Sigma$, and $d \in D$, s_0 is an initial state, and $Q_f \subseteq S$ is an acceptance condition.*

A run, R , of A_T over a tree τ abbreviated as R_τ is an infinite tree. A run, R_τ , is successful if there is a state $s_f \in Q_f$ such that R_τ hits s_f infinitely often. We say that an automaton A_T accepts a tree τ if it has a successful run R_τ . In this case we also say that an automaton A_T is not empty.

In the construction of this tree automaton, every state is labelled according to state of components (passed over from the component level automaton) and resources. In this case the transition function is not only related to the state transition of components, but is also tightly bound to the deontic logic accessibility relation. Here we expect that we would be able to specify the automaton in the normal form for CTL, SNF_{CTL} , developed in our previous papers [5]. Although we do not have a rigorous proof of this, we can anticipate that the situation here would be similar to the one in the linear-time case. Namely, in [6], it was shown that a Buchi word automaton can be represented in terms of SNF_{PLTL} , a normal form for PLTL. Similarly, we *expect that we will be able* to represent a Buchi tree automaton in terms of SNF_{CTL} . Subsequently, we enrich this representation of the automaton by deontic constraints [2] and apply a resolution based verification technique as a verification procedure.

4. Runtime Reconfiguration

4.1 Systems States

Each deployment component **must** expose a state resource property which implements the Component's Monitoring capability. To satisfy this requirement, a deployment component must contain **States** and **State Transition** elements. Additionally, a deployment component may include additional information as an opaque quantity that an external consumer may be able to process. The **Component Status** property will be exposed by every component object of a system.

We can define these properties in the XML based system architecture as:


```

<ComponentStatus>
<State>InstanceState|InitializationState|RuntimeState|
SuspendedState|FailedState|TerminatedState</State>
<LifecycleTransition>StateTransition</LifecycleTransition>
</ComponentStatus>

```

where:

| Element | Description |
|---------------------|---|
| InstanceState | State representing the presence of a component instance. |
| InitializationState | State in which a component has been properly initialized. |
| RuntimeState | Operational state. |
| SuspendedState | Operational state (in suspension). |
| FailedState | State in which the component has failed either a lifecycle operation or its operation has failed. |
| TerminatedState | State in which a component instance has been terminated. |

As the failed state may have been arrived at due to failures during many parts of the lifecycle, it is **recommended** that the component take action to ensure the services of the resources are not available while in this state, particularly if the transition occurred from the running state.

Similarly, we can map the state of resources and monitor changes through state change notifications fired by resource monitoring software implemented in the GIDE.

4.2 Formal Specification

We refer to reconfiguration as to the process through which a system halts operation under its current source specification and begins operation under a different target specification [15], and more precisely, after the deployment has taken place (dynamic reconfiguration). Some examples include the replacement of a software component by the user, or an automated healing process activated by the system itself. In either of these cases we consider the dynamic reconfiguration process as an unforeseen action at development time (known as ad-hoc reconfiguration [7]). When the system is deployed, the verification process should run continuously and the system will report back the current states for model mapping; if a reconfiguration procedure is requested or inconsistency detected, the healing process is triggered. The dynamic reconfiguration process works in a recursive way, constantly checking for update requests to the model and taking actions accordingly, enabling us to achieve an automated runtime reconfiguration through cycling deductive verification. The approach here is to specify general invariants for the infrastructure and to accept any change to the system, as long as these invariants hold. We assume that the infrastructure has some pre-defined set of norms which define the constraints for the system, in or-

der to ensure system safety, mission success, or other crucial system properties which are critical to the system.

Application scenario. We could use this type of specification to construct a normative framework for reconfiguration where a model is requested to be updated. Once an automaton at the bottom level is constructed, it feeds the upper layer automaton with the labels for the states. Then this upper layer automaton can be checked by given its presentation in the normal form with the subsequent application of the resolution procedure to the derived specification. Once this process has been carried out, we could use it for reconfiguration; when a request for reconfiguration is received, we can consider it as an update to the model, which is carried out by verifying the new specification against the system one, stopping the components in question (in essence modifying their state) and updating the model. The reconfiguration process is then left in the hands of the resource handler and the components can be started again to carry on their task with the updated model.

5. Conclusions

The need for a safe and reliable way to dynamically reconfigure systems at runtime, especially distributed, resource-depending and long-running, has led to the need for a formal way to describe and verify them before risking to take some action. In this paper we have given a novel approach to the formal specification of behaviour in GCM environments. Furthermore, by defining our automata-based approach, we have laid the grounds for a solid prototyping of such a specification system. The method introduced will be used to prevent inconsistency and suggest corrections to the system in a static and/or dynamic environment during reconfiguration procedures. Indeed, if the verification technique discovers inconsistencies in the configuration then the “healing” process is triggered: the process of “reconfiguring” of the computation tree model that conforms the protocol. As a next step, we are planning to embed all these features in a prototype plug-in for the GridComp GIDE and test it on case studies proposed by industry partners. While we have applied this framework to a GCM system, such procedure could be applied to other systems, giving the deductive reasoning a chance to assist other verification methods such as model checking by filling the gaps in those areas where these other well established methods cannot be used.

Acknowledgement

This research work was carried out under the FP6 network of excellence CoreGRID (Contract IST-2002-004265) and the FP6 research and develop-

ment project GridCOMP (Contract IST-2005-034442) funded partially by the European Commission.

References

- [1] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, N. Tonellotto. Behavioural Skeletons in GCM: Autonomic Management of Grid Components PDP '08: Proc. 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, pp. 54-63, 2008.
- [2] A. Basso and A. Bolotov. Towards GCM reconfiguration - extending specification by norms. To appear in: CoreGRID Springer Volume of the CoreGRID Workshop at Heraklion, June 2007.
- [3] A. Basukoski, V. Getov, J. Thiayalingam, S. Isaiadis. Component-Based Development Environment for Grid Systems: Design and Implementation Making Grids Work, Springer, 2008 (to appear).
- [4] T. Barros, L. Henrio, A. Cansado, E. Madelaine, M. Morel, V. Mencl and F. Plasil Extension of the Fractal ADL for the Specification of Behaviours of Distributed Components Accepted for poster presentation at the 5th Fractal Workshop (part of ECOOP'06), July 3rd, 2006, Nantes, France, Jul 2006.
- [5] A. Bolotov and M. Fisher. A Clausal Resolution Method for CTL Branching Time Temporal Logic Journal of Experimental and Theoretical Artificial Intelligence, volume 11, 1999, pages 77-93, Taylor & Francis.
- [6] A. Bolotov, C.Dixon and M. Fisher. On the Relationship between Normal Form and w -automata (with M.Fisher and C.Dixon). Journal of Logic and Computation, Volume 12, Issue 4, August 2002, pp. 561-581, Oxford University Press.
- [7] T. Batista, A. Joolia, and G. Coulson. Managing Dynamic Reconfiguration in Component-based Systems Proceedings of the European Workshop on Software Architectures, June, 2005, Springer-Velag LNCS series, Vol 3527, pp 1-18.
- [8] E. Bruneton. Fractal - Tutorial. Electronic resource: <http://fractal.objectweb.org/tutorials/fractal/index.html>. September 2003.
- [9] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In Seventh Int. Workshop on Component-Oriented Programming (WCOP02), at ECOOP 2002, Malaga, Spain, 2002.
- [10] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal component Model. Electronic resource: <http://fractal.objectweb.org/specification/fractal-specification.pdf>. February 2004.
- [11] H. Comon, M. Dauchet, R. Gilleron, C. Loding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi. Tree Automata Techniques and Applications. Available on: <http://www.grappa.univ-lille3.fr/tata>, release October, 12th 2007.
- [12] CoreGRID - The European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies. <http://www.coregrid.net/>
- [13] Basic Features of the Grid Component Model Deliverable D.PM.04, CoreGRID, March 2007.
- [14] J. Matevska-Meyer, W. Hasselbring, R.H. Reussner. Software architecture description supporting component deployment and system runtime reconfiguration. Proceedings of the Ninth International Workshop on Component-Oriented Programming, Oslo, Norway, 2004.

- [15] E.A. Strunk and J.C. Knight. Assured Reconfiguration of Embedded Real-Time Software. DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04), 2004, p. 367, IEEE Computer Society.
- [16] M.Y. Vardi. Automata-Theoretic Techniques for Temporal Reasoning From: Handbook of Modal Logic, Studies in Logic and practical Reasoning, volume 3, chapter 17, Blackbourn, Van Benthem, Wolter editors, 2006.



<http://www.springer.com/978-0-387-09454-0>

From Grids To Service and Pervasive Computing

Priol, T.; Vanneschi, M. (Eds.)

2008, XVI, 242 p., Hardcover

ISBN: 978-0-387-09454-0