View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by WestminsterResearch

WestminsterResearch
http://www.wmin.ac.uk/westminsterresearch

Temporal specification and deductive verification of a distributed component model and its environment.

Alessandro Basso
Alexander Bolotov
Vladimir Getov

School of Electronics and Computer Science

Copyright © [2009] IEEE. Reprinted from the proceedings of the Third IEEE International Conference on Secure Software Integration and Reliability Improvement, 2009 (SSIRI 2009). IEEE, pp. 379-386. ISBN 9780769537580.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Westminster's products or services. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

The WestminsterResearch online digital archive at the University of Westminster aims to make the research output of the University available to a wider audience. Copyright and Moral Rights remain with the authors and/or copyright owners.
Users are permitted to download and/or print one copy for non-commercial private study or research. Further distribution and any use of material from within this archive for profit-making enterprises or for commercial gain is strictly forbidden.

Whilst further distribution of specific materials from within this archive is forbidden, you may freely distribute the URL of the University of Westminster Eprints (http://www.wmin.ac.uk/westminsterresearch).

In case of abuse or copyright appearing without permission e-mail wattsn@wmin.ac.uk.

2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement

The 1st Workshop on Model-Based Verification & Validation Temporal Specification and Deductive Verification of a Distributed Component Model and Its Environment

Alessandro Basso, Alexander Bolotov and Vladimir Getov
School of Electronics and Computer Science
University of Westminster
Watford Road, Northwick Park
Harrow HA1 3TP, London, U.K.
(A.Basso,A.Bolotov,V.S.Getov)@westminster.ac.uk

***Abstract*—In this paper we investigate the formalisation of distributed and long-running stateful systems using our normative temporal specification framework. We analyse aspects of a component-oriented Grid system, and the benefits of having a logic-based tool to perform automated and safe dynamic reconfiguration of its components. We describe which parts of this Grid system are involved in the reconfiguration process and detail the translation procedure into a state-based formal specification. Subsequently, we apply deductive verification to test whether dynamic reconfiguration can be performed. Finally, we analyse the procedure required to update our model for reconfiguration and justify the validity and the advantages of our methodology.**

***Keywords*-Grid Component Model, Grid IDE, Automated Reconfiguration, Formal Specification, Deductive Reasoning.**

I. Introduction

Component models enable modular design of software applications that can be easily reused and combined, ensuring greater reliability. Furthermore, in distributed systems where parallel running components must be taken into consideration, the need for reliable dynamic reconfiguration is higher. In these models, components interact through interface bindings, however, there is need for a method which ensures correct composition of components and especially their interaction with the environment.

The Grid Component model (GCM) [1] is an extension of Fractal [14] built to accommodate requirements for distributed systems, in particular, those developed within and following the CoreGRID [16] project. The GCM specification defines a set of notions characterising this model, an API (Application Program Interface), an ADL (Architecture Description Language) [9] and a Deployment Descriptor file. In the GCM, when the bindings of a component is changed, this component must be stopped, more precisely, to avoid disruption to the system, when unplugging a component, such component must be stopped before severing its connections to other components. Afterwards, invocations on controller interfaces are enabled and the content of the component can be reconfigured.

The recent development of a Grid Integrated Development Environment (GIDE) based on the GCM specification [8] 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, the deployment file 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 [15] and the state information that can be retrieved through calls to the `LifeCycleController` interface for components, as well as other monitoring techniques for the environment.

The aim of this paper is to apply rigorous formal (temporal) reasoning to a component-based distributed system and the environment in which this systems performs. Thus, we apply formal specification technique based on the language of branching-time temporal logic to specify both the system and the environment. Subsequently, the deductive verification method based on the clausal resolution technique is applied to the obtained specifications. In previous research [3], [2], we have showed that this is particularly crucial during critical procedures, such as reconfiguration; and the verification of these type of models can be better achieved through the application of deductive reasoning methods as opposed to others used in similar circumstances.

The rest of this paper is organised as follows. First, in order to make the presentation self-explanatory, we describe the GCM in Section II. Then, an introduction to the normative temporal specification framework follows in Section III, while its application to the GCM and the environment is presented in Section IV. In Section V we describe the resolution-based verification technique and in Section VI we introduce the concept of dynamic reconfiguration and apply the introduced temporal reasoning framework to specify and formally verify reconfiguration scenarios. In Section VII we discuss relevant work, draw conclusions and describe

978-0-7695-3758-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSIRI.2009.61

IEEE computer society

Authorized licensed use limited to: University of Westminster. Downloaded on March 11,2010 at 05:46:52 EST from IEEE Xplore. Restrictions apply.

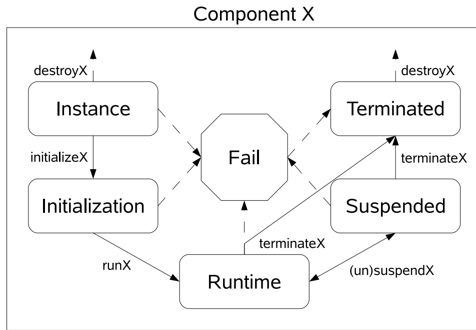

Figure 1. Component's Lifecycle States

directions for future research.

II. The Grid Component Model

In order to understand how we can formalize a grid system composed of components and resources, and clarify their places and interactions with each other, we must first have a detailed picture of the structure of such a grid system, and what are the relevant parts we must consider.

A. *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 (large components which are composites of primitive components and/or other composite components) as components that inherit the same properties and conform to state composition. 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.

B. *ADL and Deployment Descriptor*

It is well known that Architecture Description Languages (ADLs) generally cannot provide sufficient insight into the post-deployment (and thus any runtime reconfiguration) [21]; 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 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.
Similarly to the ADL, the Deployment Descriptor file details only information about the resources where the components are being deployed on (in terms of nodes, jvms, etc), disregarding any other resource which the system may require in order to function correctly. The problem has been solved by allowing to manually specify, for each component, specific requirements (or "external resources"), which can then be monitored at runtime.

III. Normative Temporal Specification Framework

In this section we introduce the temporal specification framework. It is based upon the language of the normal form ($\mathrm{SNF}_{\mathrm{CTL}}$) defined initially in [11] as the underlying language for the clausal resolution method for the computation tree logic CTL. Here we extend this setting to capture a fusion of the logic ECTL+ (extended CTL, [19]) and the deontic logic [20]. Thus, we first start with the introduction of this expressive framework and then show how $\mathrm{SNF}_{\mathrm{CTL}}$ can be extended to TDS (temporal deontic specifications) so that any formula of ECTL^+_D can be translated into a corresponding TDS.

A. ECTL^+_D *Syntax and Semantic*

In the language of ECTL^+_D, where formulae are built from the set, $Prop$, of atomic propositions $p, q, r, \ldots, p_1, q_1, r_1, \ldots, p_n, q_n, r_n, \ldots$, we use the following symbols:
classical operators: $\neg, \wedge, \Rightarrow, \vee$;
temporal operators: $\square$ – 'always in the future'; $\diamondsuit$ – 'at sometime in the future'; $\bigcirc$ – 'at the next moment in time'; $\mathcal{U}$ – 'until'; $\mathcal{W}$ – 'unless';
and path quantifiers: **A** – 'for any future path; **E** – 'for some future path.

For the deontic part we assume a set $Ag = \{a, b, c \ldots\}$ of agents (processes), which we associate with deontic modalities $\mathcal{O}_a(\varphi)$ read as 'φ is obligatory for an agent a' and $\mathcal{P}_a(\varphi)$ read as 'φ is permitted for an agent a'.

Authorized licensed use limited to: University of Westminster. Downloaded on March 11,2010 at 05:46:52 EST from IEEE Xplore. Restrictions apply.

In the syntax of ECTL_D^+ we distinguish *state* (S) and *path* (P) formulae, such that S are well formed formulae. These classes of formulae are inductively defined below (where C is a formula of classical propositional logic)

$$S ::= C|S \wedge S|S \vee S|S \Rightarrow S|\neg S|\mathbf{A}P|\mathbf{E}P|\mathcal{P}_a S|\mathcal{O}_a S$$
$$P ::= P \wedge P|P \vee P|P \Rightarrow P|\neg P|\,\Box S|\Diamond S|\bigcirc S|S\,\mathcal{U}\,S|\; S\,\mathcal{W}\,S|\,\Box\Diamond S|\Diamond\,\Box S$$

Definition 1 (literal, deontic literal): A *literal* is either p, or $\neg p$ where p is a proposition. A *deontic literal* is either $\mathcal{O}_i l$, $\neg\mathcal{O}_i l$, $\mathcal{P}_i l$, $\neg\mathcal{P}_i l$ where l is a literal and $i \in Ag$.

ECTL_D^+ **Semantics.** We first introduce the notation of tree structures, the underlying structures of time assumed for branching-time logics.

Definition 2: A *tree* is a pair (S, R), where S is a set of states and $R \subseteq S \times S$ is a relation between states of S such that $s_0 \in S$ is a unique root node, i.e. there is no state $s_i \in S$ such that $R(s_i, s_0)$; for every $s_i \in S$ there exists $s_j \in S$ such that $R(s_i, s_j)$; for every $s_i,\ s_j,\ s_k \in S$, if $R(s_i, s_k)$ and $R(s_j, s_k)$ then $s_i = s_j$.

A *path*, χ_{s_i} is a sequence of states $s_i, s_{i+1}, s_{i+2} \ldots$ such that for all $j \geq i$, $(s_j, s_{j+1}) \in R$. Let χ be a family of all paths of $\mathcal{M}$. A path $\chi_{s_0} \in \chi$ is called a *fullpath*. Let X be a family of all fullpaths of $\mathcal{M}$. Given a path χ_{s_i} and a state $s_j \in \chi_{s_i}$, $(i < j)$ we term a finite subsequence $[s_i, s_j] = s_i, s_{i+1}, \ldots, s_j$ of χ_{s_i} *a prefix* of a path χ_{s_i} and an infinite sub-sequence $s_j, s_{j+1}, s_{j+2} \ldots$ of χ_{s_i} *a suffix* of a path χ_{s_i} abbreviated $Suf(\chi_{s_i}, s_j)$.

Following [19], without loss of generality, we assume that underlying tree models are of at most countable branching.

Definition 3 (Total countable ω-tree): A countable ω-tree, τ_ω, is a tree (S, R) with the family of all fullpaths, X, which satisfies the following conditions: each fullpath is isomorphic to natural numbers; every state $s_m \in S$ has a countable number of successors; X is R-generable *[19]*, i.e. for every state $s_m \in S$, there exists $\chi_n \in X$ such that $s_m \in \chi_n$, and for every sequence $\chi_n = s_0, s_1, s_2 \ldots$ the following is true: $\chi_n \in X$ if, and only if, for every m $(1 \leq m)$, $R(s_m, s_{m+1})$.

Since in ω trees fullpaths are isomorphic to natural numbers, in the rest of the paper we will abbreviate the relation R as $\leq$.

Next, for the interpretation of deontic operators, we introduce a binary agent accessibility relation.

Definition 4 (Deontic Accessibility Relation): Given a total countable tree $\tau_\omega = (S, \leq)$, a binary agent accessibility relation $D_i \subseteq S \times S$, for each agent $i \in Ag$, satisfies the following properties: it is *serial* (for any $k \in S$, there exists $l \in S$ such that $D_i(k, l)$), *transitive* (for any $k, l, m \in S$, if $D_i(k, l)$ and $D_i(l, m)$ then $D_i(k, m)$), and *Euclidian* (for any $k, l, m \in S$, if $D_i(k, l)$ and $D_i(k, m)$ then $D_i(l, m)$).

Let $(S, \leq)$ be a total countable ω-tree with a root s_0 defined as in Def 3, X be a set of all fullpaths, $L : S \times Prop \longrightarrow \{\,\mathtt{true}, \mathbf{false}\}$ be an interpretation function mapping atomic propositional symbols to truth values at each state, and every $R_i \subseteq S \times S$ $(i \in 1, \ldots, n)$ be an agent accessibility relation defined as in Def 4. Now a model structure for interpretation of ECTL_D^+ formulae is $\mathcal{M} = \langle S, \leq, s_0, X, L, D_1, \ldots, D_n\rangle$.

Reminding that since the underlying tree structures are R-generable, they are suffix, fusion and limit closed [19], in Figure 2 we define a relation '$\models$', which evaluates well-formed ECTL_D^+ formulae at a state s_m in a model $\mathcal{M}$.

$\langle\mathcal{M}, s_m\rangle$	$\models p$	iff	$p \in L(s_m)$, for $p \in Prop$
$\langle\mathcal{M}, s_m\rangle$	$\models \mathbf{A}B$	iff	for each χ_{s_m}, $\langle\mathcal{M}, \chi_{s_m}\rangle \models B$
$\langle\mathcal{M}, s_m\rangle$	$\models \mathbf{E}B$	iff	there exists χ_{s_m} such that $\langle\mathcal{M}, \chi_{s_m}\rangle \models B$
$\langle\mathcal{M}, \chi_{s_m}\rangle$	$\models A$	iff	$\langle\mathcal{M}, s_m\rangle \models A$, for state formula A
$\langle\mathcal{M}, \chi_{s_m}\rangle$	$\models \Box B$	iff	for each $s_n \in \chi_{s_m}$, if $m \leq n$ then $\langle\mathcal{M}, Suf(\chi_{s_m}, s_n)\rangle \models B$
$\langle\mathcal{M}, \chi_{s_m}\rangle$	$\models \bigcirc B$	iff	$\langle\mathcal{M}, Suf(\chi_{s_m}, s_{m+1})\rangle \models B$
$\langle\mathcal{M}, \chi_{s_m}\rangle$	$\models A\,\mathcal{U}\,B$	iff	there exists $s_n \in \chi_{s_m}$ such that $m \leq n$ and $\langle\mathcal{M}, Suf(\chi_{s_m}, s_n)\rangle \models B$ and for each $s_k \in \chi_{s_m}$, if $m \leq k < n$ then $\langle\mathcal{M}, Suf(\chi_{s_m}, s_k)\rangle \models A$
$\langle\mathcal{M}, \chi_{s_m}\rangle$	$\models A\,\mathcal{W}\,B$	iff	$\langle\mathcal{M}, \chi_{s_m}\rangle \models \Box A$ or $\langle\mathcal{M}, \chi_{s_m}\rangle \models A\,\mathcal{U}\,B$
$\langle\mathcal{M}, s_m\rangle$	$\models \mathcal{O}_a B$	iff	for each $s_n \in S$, if $D_a(m, n)$ then $\langle\mathcal{M}, s_n\rangle \models B$
$\langle\mathcal{M}, s_m\rangle$	$\models \mathcal{P}_a B$	iff	there exists $s_n \in S$, such that $D_a(m, n)$ and $\langle\mathcal{M}, s_n\rangle \models B$

Figure 2. ECTL_D^+ semantics

Definition 5 (Satisfiability): A well-formed ECTL_D^+ formula, B, is satisfiable if, and only if, there exists a model $\mathcal{M}$ such that $\langle\mathcal{M}, s_0\rangle \models B$.

Definition 6 (Validity): A well-formed ECTL_D^+ formula, B, is valid if, and only if, it is satisfied in every possible model.

B. $\mathrm{SNF}_{\mathrm{CTL}}^D$ *Language*

To define a concept of propositional deontic temporal specification we extend a normal form defined for the logic ECTL^+, $\mathrm{SNF}_{\mathrm{CTL}}$, which was developed in [11], [13]. Recall that the core idea of the normal form is to extract from a given formula the following three types of constraints. *Initial constraints* represent information relevant to the initial moment of time, the root of a tree. *Step constraints* indicate what will happen at the successor state(s) given that some conditions are satisfied 'now'. Finally, *sometime constraints* keep track on any eventuality, again, given that some conditions are satisfied 'now'.

The $\mathrm{SNF}_{\mathrm{CTL}}^D$ language is obtained from the ECTL_D^+ language by omitting the $\mathcal{U}$ and $\mathcal{W}$ operators, and adding classically defined constants $\mathtt{true}$ and $\mathbf{false}$, and a new operator, $\mathbf{start}$ ('at the initial moment of time') defined as $\langle\mathcal{M}, s_i\rangle \models \mathbf{start}$ iff $i = 0$.

Authorized licensed use limited to: University of Westminster. Downloaded on March 11,2010 at 05:46:52 EST from IEEE Xplore. Restrictions apply.

Similarly to SNF_{CTL}, we incorporate the language for indices which is based on the set of terms
$\mathsf{IND} = \{\langle \mathsf{f} \rangle,\ \langle \mathsf{g} \rangle,\ \langle \mathsf{h} \rangle,\ \langle LC(\mathsf{f}) \rangle,\ \langle LC(\mathsf{g}) \rangle,\ \langle LC(\mathsf{h}) \rangle\ \ldots\}$, where $\mathsf{f},\ \mathsf{g},\ \mathsf{h}\ldots$ denote constants. Thus, $\mathbf{E}A_{\langle \mathsf{f} \rangle}$ means that A holds on some path labelled as $\langle \mathsf{f} \rangle$. All formulae of SNF_{CTL} of the type $P \Rightarrow \mathbf{E}\bigcirc Q$ or $P \Rightarrow \mathbf{E}\Diamond Q$, where Q is a purely classical expression, are labelled with some index.

Definition 7 (Deontic Temporal Specification - DTS): DTS is a tuple $\langle In, St, Ev, N, Lit \rangle$ where In is the set of initial constraints, St is the set of step constraints, Ev is the set of eventuality constraints, N is a set of normative expressions, and Lit is the set of literal constraints, i.e. formulae that are globally true. The structure of these constraints called *clauses*, is defined below where each α_i, β_m, γ or l_e is a literal, $\mathtt{true}$ or **false**, d_e is either a literal or a deontic literal involving the $\mathcal{O}$ or $\mathcal{P}$ operators, and $\langle \mathsf{ind} \rangle \in \mathsf{IND}$ is some index.

$\mathbf{start} \Rightarrow \bigvee_{i=1}^{k} \beta_i$	*Initial Clause*
$\bigwedge_{i=1}^{k} \alpha_i \Rightarrow \mathbf{E}\bigcirc [\bigvee_{m=1}^{n} \beta_m]_{\langle ind \rangle}$	$\mathbf{E}\bigcirc$ *clause*
$\bigwedge_{i=1}^{k} \alpha_i \Rightarrow \mathbf{A}\bigcirc [\bigvee_{m=1}^{n} \beta_m]$	$\mathbf{A}\bigcirc$ *clause*
$\bigwedge_{i=1}^{k} \alpha_i \Rightarrow \mathbf{E}\Diamond \gamma_{\langle \mathsf{LC(ind)} \rangle}$	$\mathbf{E}\Diamond$ *clause*
$\bigwedge_{i=1}^{k} \alpha_i \Rightarrow \mathbf{A}\Diamond \gamma$	$\mathbf{A}\Diamond$ *clause*
$\mathbf{true} \Rightarrow \bigvee_{e=1}^{n} d_e$	*Deontic Clause*
$\mathbf{true} \Rightarrow \bigvee_{e=1}^{n} l_e$	*Literal Clause*

IV. Formalising GCM Components and Resources

When considering what parts in the GCM can be used for formal specification, we have considered four main sections, each of which follows specific criteria and can be easily fed into our set of specification "patterns". We examine the main details below. Please note that not full specification is included for space reasons. As an example, we consider an Application (the out most component which must be activated first) which contains 4 components Comp1 (a composite component with a sub component SubComp1.1) which is the first to be started after the application is as it is the first and only component, two components CompA and CompB running in parallel from a broadcast of Comp1 (and SubComp1.1 to start in parallel with CompA or CompB), and Comp2 a component from the gathercast of CompA and CompB.

Complexity. While our initial papers on the resolution based verification of the component model model specifications [3] opened a theoretical prospect of developments in runtime reconfiguration, the complexity of the resolution based verification has raised some concerns with the feasibility of applying this method to a full scale component model. Therefore, there has been a need for complexity reduction. Unlike model checking, where the complexity lies in the specification part, namely in extracting a model, deductive reasoning 'suffers' in the verification process. One of the ways to overcome the problem is to modify the underlying specification language to obtain a lower complexity similar to the linear time resolution framework [18]. The other main and straightforward approach is to limit the actual amount specification properties considered, and therefore not needing to describe all possible combinations of states and functions of the system, as we do not need to analyse the inner working of each component, but only their stateful relations with other components and resources. In other words, the complexity of the underlying algorithm is not the length of the input formula but its structure, i.e. the way what type of subformulae are embedded in it and how.

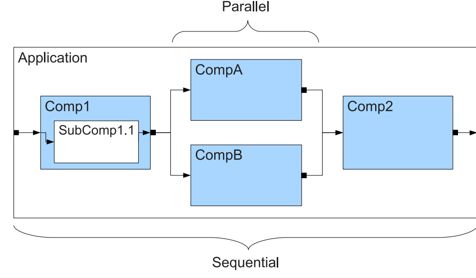

Figure 3. Sequential and Parallel Processes

A. Hierarchical Components Composition

Components in the GCM have a strict hierarchical nature. The application can then be described as:

$$\mathbf{start} \Rightarrow Application$$

and components of the application in the form of:

$$Application \Rightarrow \mathbf{A}\bigcirc Comp_1$$

$$Application \Rightarrow \mathbf{A}\Diamond Comp2$$

$$Comp1 \Rightarrow \mathbf{A}\bigcirc(SubComp_{1.1} \wedge (Comp_A \wedge Comp_B))$$

B. Inferring parallel processes from interfaces

When we consider interfaces in the GCM, we can group them in two different types: one to one, and broadcast/gathercast. In the former we have a simple connection of one server interface to a client one, while in the latter we have a single server interface which can be bound to multiple client ones.

In either case, interfaces can be very useful to determine whether the communication between components is carried out in a sequential or parallel manner. Imagine a component with a broadcast server interface (or several one to one server interfaces): we can easily assume that the components at the client side of those interfaces can be run in a parallel manner. On the other hand, a component which has only one server interface, can only run in a sequential manner with

Authorized licensed use limited to: University of Westminster. Downloaded on March 11,2010 at 05:46:52 EST from IEEE Xplore. Restrictions apply.

the component on the client interface side (see Figure: 3). Sequential specification looks like:

$$Comp1 \Rightarrow \mathbf{A}\Diamond Comp2$$

while Parallel specification looks like:

$$Comp1 \Rightarrow \mathbf{A}\bigcirc(CompA \wedge CompB)$$

When in a sequential process is easy to understand that component will be started sequentially, in a parallel process, there is no real certainty - component might be all started at the next step, or first one and then the others, or perhaps none.

C. State of resources

When considering resources, we are able to formally specify the environment thanks to information provided in the GCM deployment file as well as other metadata information gathered at development time through a development interface. Furthermore the current state of each resource can be monitored at runtime giving us a complete picture of the mapping between resources and any components that might be deployed on, or requesting the use of the resource.

D. Types of mappings

The one to one mapping. If a components is deployed to a single resource and does not require any other resource to run correctly, we define this as a one to one mapping. This is the simplest scenario.

The big component mapping. If a component is mapped to two or more resources, we define it as a big component. Such component is often a composite component, where its subcomponents have a one to one mapping with different resources, although this might not be the case as a component might have more than one resource associated to it (for ex. when a component is deployed to one resource and requires to be connected to a database which is located on another resource).

The big resource mapping. If a resource is mapped to two or more components, we define this as a big resource. This is often the case as one resource could run several virtual machine, each one running one or more components (for ex. nodes in a cluster).

E. Formalizing mappings

Independently from the type of mapping, the hierarchical structure of both components (composite and primitive components) and resources (nodes and virtual machines) is crucial in order to simplify the way we can formally describe such relations; we can in fact always translate mappings of fat components or resources as a collection of one to one mappings (sometimes with the same component or resource appearing in more than one of these mappings).
External resources are defined as:

$$Comp1 \Rightarrow \mathbf{A}\Diamond Res1$$

Deployment resources are defines as:

$$Node1 \Rightarrow Res1$$

and at runtime we can have definitions like:

$$Res1 \Rightarrow \mathbf{A}\,\Box(Comp1 \wedge Comp2)$$

F. State of components

While the states of components could have a wide spectrum of definition points (such as *initialized, started, suspended, terminated,* ... for the moment we can only consider the ones defined in the GCM - i.e. *started* and *stopped*. In a way this simplifies further the formalism by representing the specification as:

$$Comp1$$

for a started component, and:

$$\neg Comp2$$

for a stopped one.

V. Resolution Based Verification

For the specification of behaviour we can use a rich temporal framework [19] with subsequent application of either model checking or deductive reasoning as a verification technique. Model checking [17], which verifies the properties of the components against the specification, has already been tested in various circumstances, one particular application of this method been tested in [10]; it is a powerful and well established technique allowing to incorporate a number of algorithms and tools to deal even with the famous state explosion problem. However, when applied to a component system, it has one indicative drawback, namely it has an explorative nature and it cannot efficiently handle infinite state systems; in fact, model checking is used to take "snapshots" of various static states of a system, and quickly verify them, but when we consider a long running system - possible even infinite - it is easy to understand that this procedure becomes not feasible.

As a consequence, model checking cannot consider the environment in which a component system has been developed. At the same time, in building a large scale distributed system, we cannot afford any more not to take into consideration the entire infrastructure, as we have extensively analysed in our previous research [7]. Deductive methods, on the other hand, can deal with such large or even infinite state systems - as the technique has been developed precisely to solve this problem - and furthermore can be applied to reconfiguration scenarios, where we must consider future system states as a whole, and taking a series of "snapshots" would just be impractical. Resolution based verification framework for the fusion of temporal and deontic logics has been described in our previous works [2]. In [26] the original resolution method for CTL [11] has been improved by

Authorized licensed use limited to: University of Westminster. Downloaded on March 11,2010 at 05:46:52 EST from IEEE Xplore. Restrictions apply.

making the set of resolution rules more effective. This means that since in our system there is no interaction between the normative and temporal dimensions we can take this improved set of resolution rules coping with the temporal setting instead of the one we considered in [2] thus obtaining a more efficient resolution system. The correctness of the system follows from the correctness argument for both parts - temporal (as these new developments in [26] guarantee the correctness) and deontic (as shown in [2]). As a satisfactory example for deductive verification would be too long to include in this document, we refer to a simplified example we have include in our research [3] for reference.

VI. Reconfiguration

A. Static and Dynamic Configuration.

We define *static configuration* in a component model as the hierarchical structure of the components and the specific binding of interfaces which connects them. As this is a static view of the system, it cannot include the infrastructure which would complete the system - for example the resources the components will be deployed on. This process is ideal for the application of the static validation of a system, such as model checking.

Dynamic Configuration. We define *dynamic configuration* as the process in which the static configuration of the component model is applied to the infrastructure of resources, i.e. the deployment process.

Dynamic Reconfiguration We define *dynamic reconfiguration* as the process in which the mapping of components to resources varies, whether it is the removal of a component or a resource, the addition of one, the change of resources required by a specific component and so forth. This process has been historically carried out through a series of runs with model checkers at various stages of the reconfiguration, rising the complexity of the procedure. Through deductive reasoning this complexity could be reduced and applied more easily to this types of systems.

B. Formalizing Components for Reconfiguration

The need for a safe and reliable way to dynamically reconfigure systems at runtime, especially distributed, resource depending and long running, has led us to investigate a formal way to describe and verify them before risking to take some action. Utilising the extension to the specification language $\mathrm{SNF}^{D}_{\mathrm{CTL}}$, which considers deontic modalities as a way to define obligations and permissions for specific states of components and resources, we can facilitate the reconfiguration procedure. Consider the specification requests for a component X to take place of a component C as following:

$$CompA \Rightarrow \mathbf{A}\bigcirc(CompC \vee CompX)$$

and

$$ReconfigurationRequest \Rightarrow \mathbf{A}\diamondsuit CompX$$

In other words, each component is specified so that it has a possible "reconfiguration" component in place whether there will eventually be one to take its place.

C. Dynamic Reconfiguration procedures

At the abstract level used for our research, reconfiguration can take place following three levels of procedure.

At the first level some predefined event triggers the reconfiguration process where the abstract model that describes the grid system inclusive of components and resources has the new configuration introduced (for ex. a new component) and limits the old configuration where necessary. This new configuration does not require any additional change to the abstract model (we call this optimization), it can be verified and the job is then passed on to a tool to perform the reconfiguration.

At the second level, we have that the new configuration requires some additional change to the abstract model (for ex. the activation of a resource). The model is not updated and the request is passed back to the user for changes to be made before attempting the reconfiguration again. This may require additional changes to the model, but they can easily be added as an extra specification introduced before the model is verified.

At the third level, we automate the process of some cases that would normally apply to the second level (we call it automation), by making use of information embedded in the original specification which can suggest us the course of action to take to achieve reconfiguration without intervention from outside sources.

D. Model Update

Following the research in [25] we can adapt the system for model update to suit the needs of a state behaviour based distributed system. The procedure describes a way to design a model updater with built in error repair - the aim is to create a new model from the original one which ensures that only the most minimal changes are applied, by retaining as much information as possible represented in the original model. In this scenario, the model is an abstraction of our grid system inclusive of components' and resources' state behaviour; and a (satisfiable) $\mathrm{SNF}^{D}_{\mathrm{CTL}}$formula represents the new component - and all its resources connections - specified for the reconfiguration. In order to update the model to satisfy the formula, we can apply a series of operations:

(1) Adding one relation element. Given a model, its updated model is obtained by adding only one new relation element.

(2) Removing one relation element. Given a model, its updated model is obtained by removing only one existing relation element.

(3) Changing labelling function on one state. Given a model, its updated model is obtained by changing labelling

Authorized licensed use limited to: University of Westminster. Downloaded on March 11,2010 at 05:46:52 EST from IEEE Xplore. Restrictions apply.

function on a particular state.
(4) Adding one state. Given a model, its updated model is obtained by adding only one new state.
(5) Removing one isolated state. Given a model, its updated model is obtained by removing only one isolated state.

It is now easy to see how these properties can be applied in our framework. Similarly to the idea of using "patterns" during the specification of our grid system model, we can apply the operations above to create our model update formula. If we intend for example to update a component with another which performs the same function but requires an extra resource to be present, we follow this procedure: we begin from the specification provided for the original component. We identify the the property which the model does not satisfy (in this case that the extra resource is not defined in the model and it is not bound to the new component). We can then identify the two possible minimal updates (1) and (4) to add the resource and bound it to the component. After the update, we can verify that the model satisfies our formula and it has minimal change from the original. Finally we can proceed with the reconfiguration of our grid system.

E. Reconfiguration Framework

In order to give a formalisation of the reconfiguration process we adapt the approach given in [23] extending it to the usage of norms. We assume that with the notion of reconfiguration the following entities are associated:

- a set, P, of specification properties,
- a set of invariants $I \subseteq P$.
- the current system state, $S_{current}$,
- the target system state, S_{target},
- a set of norms, N.

We can now define a reconfiguration, $\mathcal{R}$, as follows.

Definition 8 (Reconfiguration): Reconfiguration is a tuple $\langle P,\ \mathcal{R},\ S_{current},\ S_{target},\ I,\ N\rangle$ which satisfy the following conditions:
- $\mathcal{R}$ commences when the current state $S_{current}$ is not operating any more and finishes before the target state to be updated, S_{target}, becomes compliant with the system.
- S_{target} is the appropriate choice for the target specification at some point of time during $\mathcal{R}$.
- Time for $\mathcal{R}$ is less or equal than the time for the transition from $S_{current}$ to S_{target}.
- The transition invariant(s), I, holds during $\mathcal{R}$.
- The norms, N, for $S_{current}$ are true at the time when $\mathcal{R}$ completes.
- The lifetime of $\mathcal{R}$ is bounded by any two occurrences of the same specification, P.

The conditions for reconfigurations can be considered as a set of restrictions, which when true allow for the model to be replaced. The reconfiguration conditions above give a clear indication which states in the model can be changed and when, while the temporal specification sets the conditions for the change and defines the acceptable states which will replace the current ones.

VII. Conclusion

To the best of our knowledge there are no analogous works in the application of the temporal reasoning techniques (with deductive verification) to the problem setting of component model and its configuration/re-configuration. Among other applications of formal methods in this area we mention formal approaches in configuration [24], [22] and the application of model checking methods [10]. However, while the former work is too general the latter approach does not consider the environment.

On the other hand, a formal approach to analyse the concept of a model update, considering various types of updates and even an attempt to design some mathematical structure of a set of updates for a given model can be found in [25]. We believe that incorporating these ideas in our work deeper would bring more light on the definition of re-configuration and on the desired automation to the process of model reconfiguration.

The formal specification procedure of component-based model and its environment introduced in this paper has been applied in reconfiguration scenarios to prevent inconsistency. Additionally, we are able to suggest possible corrections to the distributed system. Note, however, that while we have applied this framework to a GCM system, due to a generic nature of our methodology, such procedure could be applied to other systems and scenarios.

An interesting and promising prospect is seen in using the deductive reasoning 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. For example, such application can inform the model checking approach, which is very efficient when applied to a finite state system with the reasonable amount of state, but suffers from the state explosion problem when the underlying system is large.

Future works will also include a prototype development along the Grid IDE project, performance testing (and possibly some further complexity reduction analysis), as well as considering other approaches and methods which can be used and could enhance the verification procedure (such as natural deduction as a verification tool).

Acknowledgment

This research has been carried out under the European research and development project GridCOMP (Contract IST-2005-034442) funded partially by the European Commission.

Authorized licensed use limited to: University of Westminster. Downloaded on March 11,2010 at 05:46:52 EST from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Basic Features of the Grid Component Model. Deliverable D.PM.04, CoreGRID, March 2007.

[2] A. Basso and A. Bolotov Towards GCM re-configuration - extending specification by norms. Submitted to: CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments, Heraklion, 2007.

[3] A. Basso and A. Bolotov and A. Basukoski and V. Getov and L. Henrio and M. Urbanski Specification and verification of reconfiguration protocols in grid component systems. In: Proceedings of the 3rd IEEE International Conference on Intelligent Systems (IS-2006). IEEE, Los Alamitos, USA, pp. 450-455.

[4] A. Basso and A. Bolotov and V. Getov. Automata-based Formal Specification of Stateful Systems. In Proc. of Automated Reasoning Workshop, 2008.

[5] A. Basso and A. Bolotov and V. Getov. Behavioural Model of Component-based Grid Environments. From Grids To Service and Pervasive Computing, pages 19-30, Springer, 2008.

[6] A. Basso and A. Bolotov and V. Getov. State-Based Behavior Specification for GCM Systems. In Proc. of Automated Reasoning Workshop, 2009.

[7] A. Basso and A. Bolotov and V. Getov and L. Henrio. Dynamic reconfiguration of GCM components. Technical Report, CoreGRID 2008.

[8] A. Basukoski and V. Getov and J. Thiyagalingam and S. Isaiadis. Component-Based Development Environment for Grid Systems: Design and Implementation. Making Grids Work, Springer, 2008.

[9] T. Barros and L. Henrio and A. Cansado and E. Madelaine and M. Moreland 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), Nantes, France, July 2006.

[10] T. Barros and L. Henrio and E. Madelaine. Verification of Distributed Hierarchical Components. In Proc. of the International Workshop on Formal Aspects of Component Software (FACS'05). Electronic Notes in Theor. Computer Sci. 160. pp. 41-55 (ENTCS), 2005.

[11] A. Bolotov. Clausal Resolution for Branching-Time Temporal Logic. PhD thesis, Department of Computing and Mathematics, The Manchester Metropolitan University, 2000.

[12] A. Bolotov and 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.

[13] 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.

[14] E. Bruneton and 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.

[15] E. Bruneton and T. Coupaye and J.B. Stefani. The Fractal component Model. Electronic resource: http://fractal.objectweb.org/specification/fractal-specification.pdf. February 2004.

[16] 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/

[17] E. M. Clarke and A. Fehnker and S. Jha and H. Veith. Temporal Logic Model Checking., Handbook of Networked and Embedded Control Systems, 2005, pages 539-558.

[18] C. Dixon and M. Fisher and B. Konev. Tractable Temporal Reasoning. In Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), pages 318-323, January 6-12th 2007, Hyderabad, India.

[19] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics., pages 996–1072. Elsevier, 1990.

[20] A. Lomuscio and B. Wozna. A complete and decidable axiomatisation for deontic interpreted systems. In DEON, volume 4048 of Lecture Notes in Computer Science, pages 238–254. Springer, 2006.

[21] J. Matevska-Meyer and W. Hasselbring and 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.

[22] M. Niamanesh and N.F. Nobakht and R. Jalili and F. H. Dehkordi. On Validity Assurance of Dynamic Reconfiguration for Component-based Programs Electronic Notes in Theoretical Computer Science Volume 159, 24 May 2006, Pages 227-239 Proceedings of the First IPM International Workshop on Foundations of Software Engineering (FSEN 2005)

[23] 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.

[24] M. Wermelinger and J. L. Fiadeiro. A Graph Transformation Approach to Software Architecture Reconfiguration. Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems GraTra2000.

[25] Y. Zhang and Y. Ding. CTL Model Update for System Modifications. Journal of Artificial Intelligence Research, 2008; 31:113-155.

[26] L. Zhang and U. Hustadt and C. Dixon. First-order resolution for CTL. Technical Report ULCS-08-010, Department of Computer Science, University of Liverpool, 2008.

Authorized licensed use limited to: University of Westminster. Downloaded on March 11,2010 at 05:46:52 EST from IEEE Xplore. Restrictions apply.