*Моделирование и анализ информационных систем.* Т. 23, №6 (2016), с. 804–825
*Modeling and Analysis of Information Systems.* Vol. 23, No 6 (2016), pp. 804–825



**DOI:** 10.18255/1818-1015-2016-6-804-825

УДК 519.987

# Реконфигурирование компонентно-ориентированных систем на базе графовых грамматик

**Кушнаренко О. Б., Вебер Ж-Ф.**

*получена 15 октября 2016*

**Аннотация.** Динамические реконфигурирования могут изменять архитектуру компонентно-ориентированных систем, не подвергаясь никакому системному простою. В этом контексте основной вклад данной статьи – доказательство результатов корректности реконфигурирования систем, используя графовые грамматики. В этой статье предложены новые охраняемые реконфигурирования на базе логики Хоара, которые построены на основе примитивных операций по реконфигурированию и включают последовательности реконфигурирований, альтернативные и повторяющиеся конструкции, сохраняя при этом непротиворечивость конфигураций. Практический вклад состоит в описании имплементации компонентно-ориентированной модели, используя программный инструмент *GROOVE* для преобразования графов. После обогащения модели интерпретированными конфигурациями и реконфигурированиями, совместимого с непротиворечивостью, отношение симуляции используется для доказательства корректности имплементации, выполненной под *GROOVE*. Эта имплементация иллюстрирована на примере многоуровневого облачно-ориентированного приложения.

**Ключевые слова:** компонентно-ориентированные системы, динамическое реконфигурирование, непротиворечивость, соответствие, реализация, *GROOVE*

**Для цитирования:** Кушнаренко О. Б., Вебер Ж-Ф., "Реконфигурирование компонентно-ориентированных систем на базе графовых грамматик", *Моделирование и анализ информационных систем*, **23**:6 (2016), 804–825.

**Об авторах:**
Кушнаренко Ольга Борисовна, orcid.org/0000-0003-1482-9015, профессор информатики, доктор,
Университет Бургундия Франш-Комтэ,
ул. Рут де Грей, 16, г. Безансон, 25000 Франция, e-mail: olga.kouchnarenko@univ-fcomte.fr

Вебер Жан-Франсуа, аспирант,
Университет Бургундия Франш-Комтэ,
ул. Рут де Грей, 16, г. Безансон, 25000 Франция, e-mail: jfweber@femto-st.fr

**Благодарности:**
Работа выполнена при финансовой поддержке французской целевой программы «Labex ACTION, ANR-11-LABEX-0001-01».

## 1. Введение

Динамические реконфигурирования, которые изменяют архитектуру самонастраивающихся [1] компонентно-ориентированных систем, не подвергаясь никакому системному времени простоя, должны происходить не только при подходящих обстоятельствах, но также должны сохранять непротиворечивость систем. В то время,

как первое может быть обеспечено политиками адаптации, второе непосредственно связано с определением реконфигурирований.

Для определения свойств поведения компонентно-ориентированных систем темпоральная логика линейного времени, основанная на работе Двайера над шаблонами и рамками их применения [2], была предложена в [3]. Эта логика является расширением JML-спецификаций временными шаблонами, введенными в [4]. Эта логика, названная FTPL[1], используется, чтобы инициировать политики адаптации в [5]. Кроме того, децентрализованное вычисление свойств FTPL по наборам компонентов было изучено в [6]. Что касается условий на непротиворечивость компонентно-ориентированных систем, определенных в [7], доказать их сохранение для систем под наблюдением было непросто, главным образом из-за отсутствия точной семантики для примитивных операций по реконфигурированию.

Для более сложных реконфигурирований, составленных из примитивных операций в виде последовательностей, повторений или выборов, предварительные условия реконфигурирований и их выходные условия должным быть выражены точным и кратким способом. Поэтому в данной работе используется понятие *самого слабого предварительного условия*, введенное в [8], чтобы выразить непримитивные *охраняемые реконфигурирования*.

Далее, используя программный инструмент *GROOVE* преобразования графов [9], предложена реализация для выполнения динамического реконфигурирования над моделями компонентно-ориентированных систем, использующая графовые грамматики. Это практическое внедрение позволяет нам не только имитировать процесс реконфигурирования системы, но также генерировать комбинации возможных реконфигурирований. Так как данная работа имеет целью формализацию реконфигурирования на базе графовых грамматик, основной результат этой работы состоит в доказательстве корректности интерпретируемых систем по отношению к нашей модели реконфигурирования, используя для их выполнения продукции над графами. Это также демонстрирует правильность нашей реализации.

Отметим, что эта работа мотивирована приложениями в многочисленных моделях и платформах, поддерживающих разработку компонентов вместе с их мониторами и контроллерами, такими как, например, Fractal [10], CSP‖B [11], Frascati [12] и т.д.

Статья организована следующим образом. Пример окружения для приема облачно-ориентированных многоуровневых приложений, рассматриваемый в качестве компонентно-ориентированной системы, представлен в разделе 2. Необходимая информация о нашей компонентно-ориентированной модели реконфигурирования, а также элементы ее операционной семантики даны в разделе 3. Реализация модели, использующая программный инструмент *GROOVE* для выражения реконфигурирований посредством графовых грамматик, представлена на базе примера в разделе 4. Наконец, результаты корректности даны в разделе 5. Библиографические заметки и заключение содержатся в разделе 6.

[1]FTPL обозначает TPL (Temporal Pattern Language) с приставкой 'F', чтобы обозначить ее связь к компонентам Fractal и к условиям первого порядка для их целостности.

## 2. Пример системы с компонентами: многоуровневое облачно-ориентированное приложение

Интернет-провайдеры и телекоммуникационные операторы склонны все больше и больше определять себя поставщиками "облачной" инфраструктуры. В этом контексте автоматизация программного обеспечения и виртуальной установки и конфигурации оборудования является важной задачей. Однако этого недостаточно, чтобы приложение было облачно-применяемым; оно должно быть масштабируемым, и механизмы для этого должны быть интегрированы в систему управления облаком.

Рассмотрим типичное веб-приложение с компонентами трёхуровневых приложений, использующее клиентскую часть веб-сервера, сервер приложений промежуточного программного обеспечения и серверное приложение для работы с данными. На рис. 1 изображена управляемая виртуальная машина *VM*, принимающая такое приложение.

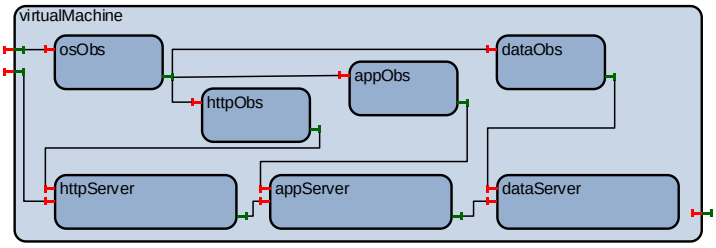



Рис. 1: Управляемая виртуальная машина с компонентами трёхуровневых приложений
Fig. 1. Managed virtual machine with three-tier application compoments

VM представлена составным компонентом *virtualMachine* с подкомпонентами *httpServer*, *appServer* и *dataServer* служб приложения. Подкомпоненты служб имеют два предоставляемых интерфейса: один для предоставления услуги, а другой для её мониторинга.

Кроме того, VM на рис. 1 также содержит четыре *наблюдателя*, которые являются подкомпонентами для мониторинга служб. Подкомпонент *osObs* используется для мониторинга операционной системы VM. Также он связан с подкомпонентами *httpObs*, *appObs* и *dataObs*, используемыми соответственно для мониторинга служб *httpServer*, *appServer* и *dataServer* подкомпонентов. Наконец, отметим, что у самого составного компонента VM есть два предоставляемых интерфейса: один для предоставления сервисов и второй для мониторинга.

На рисунке 2 изображена *Облачная среда cloudEnv*, содержащая VM для использования в целях разработки (*vmDev*). В ней имеются трёхуровневые приложения без мониторинга; назовем такую VM *неконтролируемой*. Три другие VM – *управляемые*, и каждая содержит часть приложения. Читатель может отметить, что каждая из управляемых VM содержит только наблюдателей для мониторинга операционной системы и типа предоставляемой услуги. У облачной среды есть три предоставляемых интерфейса: два, чтобы предоставить услугу, будь то версия разработки или нет, а третий для мониторинга, подключенный к подкомпоненту *monitorObs*, связанному со всеми контролируемыми интерфейсами управляемых VM.

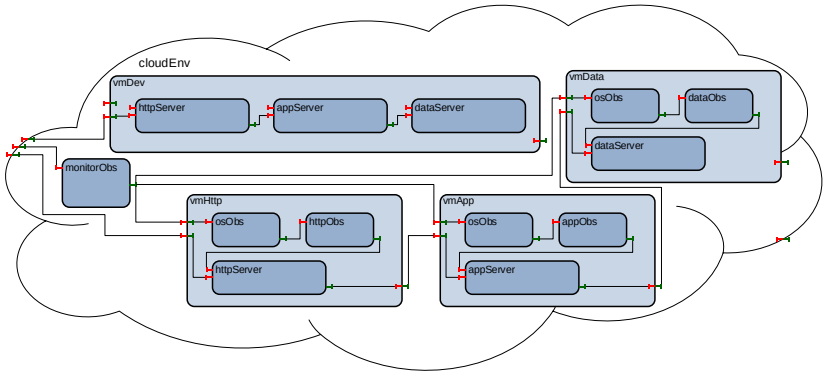



Рис. 2: Пример облачной среды
Fig. 2. Cloud environment example

Поставщик "облачной" инфраструктуры должен быть в состоянии обеспечить по требованию одну или несколько VM, сконфигурированных с правильными компонентами службы и соответствующим мониторингом. В этом контексте мы изучаем подготовку к эксплуатации VM, показанной на рис. 1. В зависимости от предоставляемых услуг и состояния мониторинга (для управляемой или неуправляемой VM) должны быть добавлены необходимые компоненты. Во время жизненного цикла VM могут произойти некоторые изменения конфигурации; мы рассматриваем их как реконфигурирование компонентно-ориентированной системы.

# 3. Компонентно-ориентированные системы: семантическая модель

## 3.1. Конфигурации и реконфигурации

Компонентные модели могут быть очень разнородными. В большинстве из них компоненты программного обеспечения рассматриваются как черные ящики с полностью специфицированными интерфейсами (или как серые ящики, если некоторые их внутренние особенности видимы). Определения компонентов и их интерфейсы используются для описания взаимодействий и сложного поведения. В этом разделе мы возвращаемся к модели реконфигурирования архитектуры компонентно-ориентированных систем, введенной в [13, 7]. В общем случае конфигурация системы – это определение элементов, из которых состоит система, в то время как реконфигурирование может быть рассмотрено как переход от одной конфигурации к другой.

Следуя за [13], конфигурация определена как множество архитектурных элементов (компоненты, предоставляемые и требуемые интерфейсы, параметры) и отношений, чтобы структурировать и связать их.

**Определение 1** (Конфигурация). *Конфигурацией c называется набор* $\langle Elem, Rel \rangle$, *где*

- $Elem = Components \uplus Interfaces \uplus Parameters \uplus Types$ – *множество архитектурных элементов, таких что*

– $Components$ – *непустое множество компонентов;*

– $Interfaces = RequiredInts \uplus ProvidedInts$ – *конечное множество предоставляемых и требуемых интерфейсов;*

– $Parameters$ – *конечное множество параметров компонентов;*

– $Types = ITypes \uplus PTypes$ – *конечное множество типов интерфейсов и типов данных для параметров;*

- $Rel = \left\{ \begin{array}{l} Container \uplus ContainerType \uplus Contingency \\ \uplus\, Parent \uplus Depth \uplus Binding \uplus Delegate \uplus State \uplus Value \end{array} \right.$

*– множество архитектурных отношений, которые связывают элементы, такие что*

– $Container : Interfaces \uplus Parameters \to Components$ – всюду определенная функция, *дающая компоненту с рассматриваемым интерфейсом или с рассматриваемым параметром;*

– $ContainerType : Interfaces \uplus Parameters \to Types$ – всюду определенная функция, *дающая тип каждого интерфейса или параметра;*

– $Contingency : RequiredInts \to \{mandatory, optional\}$ – всюду определенная функция, *указывающая, каким (mandatory или optional) является каждый требуемый интерфейс;*

– $Parent \subseteq Components \times Components$ – отношение, *связывающее подкомпонент с соответствующим составным компонентом* [2]*;*

– $Depth : Components \to \mathbb{N}$ – всюду определенная функция, *определяющая глубину компонентов;*

– $Binding : ProvidedInts \to RequiredInts$ – частичная функция, *связывающая вместе предоставляемый и требуемый интерфейсы;*

– $Delegate : Interfaces \to Interfaces$ – частичная функция *для выражения делегируемых связей;*

– $State : Components \to \{started, stopped\}$ – всюду определенная функция, *дающая статус инициированных компонентов;*

– $Value : Parameters \to \{t | t \in PType\}$ – всюду определенная функция, *дающая текущую величину каждого параметра.*

Определим также множество $CP$ свойств конфигурации, которые являются ограничениями на архитектурные элементы и отношения между ними. Эти свойства определяются формулами логики первого порядка [14]. Интерпретация функций, отношений и предикатов над $Elem$ дается согласно базовым определениям в [14] и определению 1 (см. [7] для дополнительной информации).

Пусть $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ – множество конфигураций. *Интерпретация* $l : \mathcal{C} \to CP$ дает самую широкую конъюнкцию $cp \in CP$, которая является истинной на $c \in \mathcal{C}$.

[2] Для каждого $(p, q) \in Parent$ $p$ называется подкомпонентом $q$. Разделяемые компоненты (подкомпоненты с множественным вложением в составные компонеты) могут иметь более одного родителя.

Мы говорим, что конфигурация $c = \langle Elem, Rel \rangle$ удовлетворяет формуле $cp \in CP$, когда $l(c) \Rightarrow cp$; в этом случае $cp$ действительна на $c$, иначе $c$ не удовлетворяет $cp$.

Среди свойств конфигураций архитектурные *условия на непротиворечивость* $CC$ в таблице 1 выражают требования на составление компонентов. Эти требования являются общими для всех компонентно-ориентированных архитектур [7]. Неформально,

- компонент *поставляет*, по крайней мере, один предоставляемый интерфейс (**CC.1**);
- сложные компоненты не имеют параметров (**CC.2**);
- подкомпонент не должен включать своего собственного родителя (**CC.3**);
- два связанных интерфейса должны быть одного и того же типа (**CC.4**), и содержащие их компоненты являются подкомпонентами одного и того же составного компонента (**CC.5**);
- связывая два интерфейса, необходимо гарантировать, что они не были еще вовлечены в делегированную связь (**CC.6**); точно так же, устанавливая делегированную связь между двумя интерфейсами, спецификатор должен гарантировать, что они не были еще связаны между собой (**CC.7**);
- предоставляемый (соответственно требуемый) интерфейс подкомпонента делегируется самое большее одному предоставляемому (соотв. требуемому) интерфейсу родительского компонента (**CC.8**), (**CC.9**) и (**CC.11**); интерфейсы, вовлеченные в делегацию, должны быть одного и того же типа (**CC.10**);
- компонент находится в состоянии *started*, только если его обязательные требуемые интерфейсы связаны или делегированы (**CC.12**).

**Определение 2** (Непротиворечивая конфигурация)**.** *Пусть* $c = \langle Elem, Rel \rangle$ *– конфигурация, и* $CC$ *– условия на непротиворечивость. Конфигурация* $c$ *называется* непротиворечивой*, что обозначено* $\texttt{consistent}(c)$*, если* $l(c) \Rightarrow CC$*. Определим* $\texttt{consistent}(\mathcal{C})$*, когда* $\forall c \in \mathcal{C}.\texttt{consistent}(c)$*.*

Дополнительная информация о непротиворечивых конфигурациях может быть найдена в [7].

## 3.2. Операционная семантика

Реконфигурирования заставляют компонентно-ориентированную архитектуру изменяться динамично. Они состоят из примитивных операций, таких как инициирование/разрушение (*new*/*destroy*) компонентов; добавление/удаление (*add*/*remove*) компонентов; связывание/развязывание (*bind*/*inbind*) интерфейсов; старт/остановка (*start*/*stop*) компонентов; присваивание значений параметрам компонентов (*update*). Эти примитивные операции удовлетворяют pre/post-предикаты. Например, прежде чем добавить $comp_1$ подкомпонента к составному $comp_2$, нужно проверить,

Таблица 1: Условия на непротиворечивость

Table 1: Consistency constraints

$$\forall c.(c \in Components \Rightarrow (\exists ip.(ip \in ProvidedInts \wedge Container(ip) = c))) \quad \textbf{(CC.1)}$$

$$\forall c, c' \in Components.(c \neq c' \wedge (c, c') \in Parent \Rightarrow \forall\, p.(p \in Parameters \Rightarrow Container(p) \neq c')) \quad \textbf{(CC.2)}$$

$$\forall c, c' \in Components.((c, c') \in Parent \Rightarrow Depth(c') < Depth(c)) \quad \textbf{(CC.3)}$$

$$\begin{matrix}\forall ip \in ProvidedInts, \\ \forall ir \in RequiredInts\end{matrix} \cdot \left( Binding(ip) = ir \Rightarrow \begin{matrix} ContainerType(ip) = ContainerType(ir) \\ \wedge Container(ip) \neq Container(ir) \end{matrix} \right) \quad \textbf{(CC.4)}$$

$$\begin{matrix}\forall ip \in ProvidedInts, \\ \forall ir \in RequiredInts\end{matrix} \cdot \left( Binding(ip) = ir \Rightarrow \exists c \in Components. \left( \begin{matrix} (Container(ip), c) \in Parent \\ \wedge (Container(ir), c) \in Parent \end{matrix} \right) \right) \quad \textbf{(CC.5)}$$

$$\begin{matrix}\forall ip \in ProvidedInts, \\ \forall ir \in RequiredInts\end{matrix} \cdot \left( Binding(ip) = ir \Rightarrow \begin{matrix} ip \notin dom(Delegate) \\ \wedge ir \notin dom(Delegate) \end{matrix} \right) \quad \textbf{(CC.6)}$$

$$\forall i, i' \in Interfaces. \left( Delegate(i) = i' \Rightarrow \begin{matrix} i \notin dom(Binding) \\ \wedge i \notin codom(Binding)) \end{matrix} \right) \quad \textbf{(CC.7)}$$

$$\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in ProvidedInts \Rightarrow i' \in ProvidedInts) \quad \textbf{(CC.8)}$$

$$\forall i, i' \in Interfaces.(Delegate(i) = i' \wedge i \in RequiredInts \Rightarrow i' \in RequiredInts) \quad \textbf{(CC.9)}$$

$$\forall i, i' \in Interfaces. \left( Delegate(i) = i' \Rightarrow \begin{matrix} ContainerType(i) = ContainerType(i') \\ \wedge\ (Container(i), Container(i')) \in Parent \end{matrix} \right) \quad \textbf{(CC.10)}$$

$$\forall i, i', i" \in Interfaces. \left( \begin{matrix} (Delegate(i) = i' \wedge Delegate(i) = i" \Rightarrow i' = i") \\ \wedge (Delegate(i) = i" \wedge Delegate(i') = i" \Rightarrow i = i') \end{matrix} \right) \quad \textbf{(CC.11)}$$

$$\forall ir \in RequiredInts. \left( \begin{matrix} State(Container(ir)) = started \\ \wedge Contingency(ir) = mandatory \end{matrix} \Rightarrow \exists i \in Interfaces. \left( \begin{matrix} Binding(i) = ir \\ \vee Delegate(i) = ir \\ \vee Delegate(ir) = i \end{matrix} \right) \right) \quad \textbf{(CC.12)}$$

Таблица 2: Предварительные условия примитивной операции *add*

Table 2: Preconditions of the *add* primitive reconfiguration operation

$$comp_1, comp_2 \in Components \quad (2) \qquad (comp_2, comp_1) \notin Parent^+ \quad (4)$$

$$comp_1 \neq comp_2 \quad (3) \qquad \forall p \in Parameters, (p, comp_2) \notin Container \quad (5)$$

как указано в таблице 2, следующее: *a*) $comp_1$ и $comp_2$ существуют (2), и они различны (3), *b*) $comp_2$ не является потомком $comp_1$ (4), и *c*) $comp_2$ не имеет параметров (5). Когда эти предварительные условия удовлетворены, выходное условие состоит в добавлении $(comp_1, comp_2)$ к отношению $Parent$, формально $R_{add} = Parent \cup \{(comp_1, comp_2)\}$ (1).

Следуя за основанной на предикатах семантикой базовых конструкций языков программирования [15], рассмотрим операцию реконфигурирования *ope* и две конфигурации $c$ и $c'$, такие что переход между $c$ и $c'$ выполняется с использованием *ope*. Пусть $R$ – условия на конфигурации системы под наблюдением, тогда $wp(ope, R)$ обозначает, как в [8], *самое слабое предварительное условие* на конфигурацию $c$, такое что активация *ope* может произойти, и тогда она гарантированно ведет к $c'$, удовлетворяющей выходному условию $R$. Формально, в нашем случае, если $l(c) \Rightarrow wp(ope, R)$ и $c \xrightarrow{ope} c'$, тогда $l(c') \Rightarrow R$. Поэтому для примитивной операции $add(,)$ самым слабым предварительным условием $wp(add, R_{add})$ является конъюнкция предварительных условий (2) – (5).

На базе [8] и используя такие же обозначения, таблица 3 приводит грамматику для *охраняемых реконфигурирований*, имеющую аксиомой <guarded configuration>. Пусть <*ope*> представляет примитивную операцию по реконфигурированию, так-

Таблица 3: Грамматика охраняемых реконфигурирований

Table 3: Guarded reconfigurations grammar

| | | |
|---|---|---|
| <guarded reconfiguration> | ::= | <guard> → <guarded list> |
| <guard> | ::= | <boolean expression> |
| <guarded list> | ::= | <statement>{; <statement> } |
| <guarded reconfiguration set> | ::= | <guarded reconfiguration>{[] <guarded reconfiguration> } |
| <alternative construct> | ::= | **if** <guarded reconfiguration set> **fi** |
| <repetitive construct> | ::= | **do** <guarded reconfiguration set> **od** |
| <statement> | ::= | <alternative construct> \| <repetitive construct> \| <*ope*> |

же называемую *примитивным утверждением*. Расширим множество примитивных операций по реконфигурированию операцией *skip*, которая не вызывает изменения на данной конфигурации. Следовательно, для любого выходного условия $R$, верно $wp(skip, R) = R$. Дополнительно, как в [8], семантика оператора ";" определена как $wp(S_1; S_2, R) = wp(S_1, wp(S_2, R))$, где $S_1$ и $S_2$ – утверждения.

Если множество охраняемых реконфигурирований содержит более одного элемента, они разделены [] [3]. Чтобы представить семантику альтернативной конструкции, пусть $IF$ обозначает **if** $B_1 \to S_1 [] \ldots [] B_n \to S_n$ **fi**, и $BB$ обозначает $(\exists i : 1 \leq i \leq n : B_i)$, тогда $wp(IF, R) = BB \wedge (\forall i : 1 \leq i \leq n : B_i \Rightarrow wp(S_i, R))$. Для периодически повторяющейся конструкции $DO$ обозначает **do** $B_1 \to S_1 [] \ldots [] B_n \to S_n$ **do**. Пусть $H_0(R) = R \wedge \neg BB$ и для $k > 0$, $H_k(R) = wp(IF, H_{k-1}(R)) \vee H_0(R)$, тогда $wp(DO, R) = \exists k : k \geq 0 : H_k(R)$. Интуитивно, $H_k(R)$ является самым слабым предварительным условием, обеспечивающим завершение после не больше чем $k$ выборов из охраняемого списка, оставляя систему в конфигурации, удовлетворяющей $R$. Пусть $\mathcal{R}_{run} = \mathcal{R} \cup \{run\}$ – множество операций, где $\mathcal{R}$ – конечное множество охраняемых реконфигурирований, инициированное относительно рассматриваемой системы, и $run$ – название универсального действия, представляющего все текущие операции[4] компонентно-ориентированной системы.

**Определение 3** (Модель реконфигурирования)**.** *Операционная семантика компонентно-ориентированной системы определяется маркированной системой с переходами* $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \to, l \rangle$, *где* $\mathcal{C} = \{c, c_1, c_2, \ldots\}$ *– множество конфигураций,* $\mathcal{C}^0 \subseteq \mathcal{C}$ *– множество начальных конфигураций,* $\to \subseteq \mathcal{C} \times \mathcal{R}_{run} \times \mathcal{C}$ *– отношение реконфигурирования, подчиняющееся предикатам* $wp()$, *и* $l : \mathcal{C} \to CP$ *– всюду определенная функция интерпретации конфигураций.*

Обозначим $c \stackrel{ope}{\to} c'$, когда $(c, ope, c') \in \to$. Для модели $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \to, l \rangle$, определим путь $\sigma$ в $S$ как последовательность конфигураций $c_0, c_1, c_2, \ldots$, таких что $\forall i \geq 0$. $\exists\, ope_i \in \mathcal{R}_{run}.(c_i \stackrel{ope_i}{\to} c_{i+1})$. Пусть $\Sigma$ обозначают множество путей, а $\Sigma^f$ $(\subseteq \Sigma)$ множество конечных путей. Определим выполнение как путь $\sigma$ в $\Sigma$, такой что $\sigma(0) \in \mathcal{C}^0$. Пусть $\sigma(i)$ обозначает $i$-ю конфигурацию $\sigma$. Обозначение $\sigma_i$ используется для суффикса $\sigma(i), \sigma(i+1), \ldots$, и $\sigma_i^j$ обозначает отрезок пути $\sigma(i), \sigma(i+1), \ldots, \sigma(j-1), \sigma(j)$. Конфигурация $c'$ достижима из $c$, когда существует путь $\sigma = c_0, c_1, \ldots, c_n$ в $\Sigma^f$, такой что $c = c_0$ и $c' = c_n$ с $n \geq 0$. Пусть $c$ – конфигурация, тогда $reach(c)$ обозначает

[3]Как и в [8], порядок, в котором появляются охраняемые реконфигурирования, семантически не важен.

[4]Нормальное функционирование различных компонентов также изменяет архитектуру, например, изменяя значения параметров или останавливая компоненты.

множество всех конфигураций, достижимых из $c$. Это понятие может быть поднято с конфигураций на множество конфигураций: $reach(\mathcal{C}) = \{reach(c) \mid c \in \mathcal{C}\}$.

**Утверждение 1** (Сохранение непротиворечивости)**.** *Пусть* $\mathcal{C}^0 \subseteq \mathcal{C}$*. Тогда* $\texttt{consistent}(\mathcal{C}^0)$ *влечет* $\texttt{consistent}(reach(\mathcal{C}^0))$*.*

*Набросок.* Установим сначала, что каждая примитивная операция *ope* сохраняет непротиворечивость конфигурации. Это означает, что для выходного условия $R$ операции *ope* верно $CC \wedge wp(ope, R) = wp(ope, CC \wedge R)$. Докажем этот результат для *add(,)*, доказательство для других примитивных операций является схожим. Пусть $\texttt{consistent}\,(c)$, и предварительные условия для $add(,)$ выполнены на $c$. Тогда переход $c \xrightarrow{add} c'$ должен вести к непротиворечивой конфигурации $c'$ ($\texttt{consistent}\,(c')$), такой что выходные условия для $add(,)$ также удовлетворяют условиям на непротиворечивость в таблице 1. Формально, $(l(c) \Rightarrow CC \wedge wp(add, R_{add})) \wedge (c \xrightarrow{add}, c') \Rightarrow (l(c') \Rightarrow CC \wedge R_{add})$. Действительно, так как отношение *Parent* из выходного условия (1) не вовлечено в (**CC.1**), (**CC.4**) – (**CC.9**), (**CC.11**), и (**CC.12**), эти условия также выполнены для $c'$. Для остающихся условий мы имеем:

(**CC.2**): Предварительное условие (5) из таблицы 2 гарантирует, что у родительского компонента $comp_2$ нет параметров, (**CC.2**) выполнено на $c'$ с $(comp_1, comp_2)$, добавленной к *Parent* (cf. (1));

(**CC.3**): Предварительное условие (4) в таблице 2 означает, что $comp_2$ не может быть потомком $comp_1$, таким образом предотвращая цикл в отношении *Parent* для $c'$, когда $comp_2$ становится родителем $comp_1$;

(**CC.10**): Существуют два случая: либо делегированная связь между интерфейсами $comp_1$ и $comp_2$ на $c$ до выполнения операции $add(,)$ уже существовала, либо нет. В последнем случае ограничение (**CC.10**) тривиально выполнено на $c'$. В первом случае, поскольку $\texttt{consistent}\,(c)$, отношение *Parent* уже содержало $(comp_1, comp_2)$ с интерфейсами правильного типа, и применение $add(,)$ не изменяет типы и отношение, поэтому условие выполнено на $c'$.

Пусть $c \in reache(\mathcal{C}^0)$. По определению существует $c_0 \in \mathcal{C}^0$ и последовательность операций из $\mathcal{R}_{run}$, чтобы в конечном счете достигнуть $c$. По определению также существует последовательность примитивных операций $ope_0, ope_1, \ldots, ope_{n-1}$ и ряд промежуточных конфигураций $\mathcal{C}' = \{c_1, c_2, \ldots, c_{n-1}\}$[5], таких что $c_0 \xrightarrow{ope_0} c_1$, $c_1 \xrightarrow{ope_1} c_2$, $\ldots$, $c_{n-1} \xrightarrow{ope_{n-1}} c$, и для $0 \leq i \leq n-1$, $c_i$ (соотв. $c_{i+1}$), выполняют предварительные условия (соотв. выходные условия) $ope_i$ (с $c_n$, обозначающей $c$). Действительно, если бы эта последовательность примитивных операций или $\mathcal{C}'$ не существовали, $c$ не была бы достижима ни из какой конфигурации в $\mathcal{C}^0$.

Теперь докажем, что охраняемое реконфигурирование, имеющее последовательность примитивных утверждений в своем охраняемом списке, сохраняет непротиворечивость. Пусть $gl_n$ – охраняемый список, составленный из $n \geq 0$ примитивных

[5]Заметим, что $\mathcal{C}'$ не обязательно является подмножеством $\mathcal{C}$. Например, если каждая операция в $\mathcal{R}$ является последовательностью двух примитивных операций, тогда промежуточные конфигурации не принадлежат $\mathcal{C}$, и $\mathcal{C}' \not\subset \mathcal{C}$.

операций, т.е., $gl_n = ope_0; ope_1; \ldots; ope_n$, с $R_i$ и $R_{i+1}$, являющихся соответственно предварительными и выходными условиями $ope_i$. Обозначим $CC_i = CC \wedge R_i$. Докажем по индукции на $n$, что $CC_0 = wp(gl_n, CC_{n+1})$. Для $n = 0$ у нас есть $gl_n = ope_0$ и $CC_0 = wp(gl_0, CC_1)$. Рассмотрим теперь $gl_{n+1} = gl_n; ope_{n+1}$; мы имеем $wp(gl_{n+1}, CC_{n+2}) = wp(gl_n, wp(ope_{n+1}, CC_{n+2}))$. Так как $CC_0 = wp(gl_n, CC_{n+1})$ и $CC_{n+1} = wp(ope_{n+1}, CC_{n+2})$, мы имеем, по определению [8], $CC_0 = wp(gl_n, CC_{n+1}) = wp(gl_n, wp(ope_{n+1}, CC_{n+2}))$.

Используя гипотезу на предварительные и выходные условия утверждений, в приложении 6.2. доказано, что охраняемые реконфигурирования с непримитивным утверждением, использующим только примитивные утверждения ($G \to$ **fi** $grs$ **fi** или $G \to$ **do** $grs$ **od**, где $grs$ обозначает $B_0 \to ope_0 [\!] B_1 \to ope_1 [\!] \ldots [\!] B_n \to ope_n$), также предохраняют непротиворечивость.

Таким образом, с тем же рассуждением, рассматривая непримитивные заявления вместо примитивных и используя только гипотезу на предварительные и выходные условия утверждений, мы можем доказать, что непротиворечивость сохранена *a*) для охраняемых реконфигурирований, имеющих охраняемый список, составленный из последовательности непримитивных утверждений ($G \to S_0; S_1; \ldots; S_n$) и *b*) для охраняемых реконфигурирований, имеющих утверждение в качестве охраняемого списка ($G \to$ **fi** $grs$ **fi** или $G \to$ **do** $grs$ **od**, где $grs$ обозначает $B_0 \to S_0 [\!] B_1 \to S_1 [\!] \ldots [\!] B_n \to S_n$). □

# 4. Реализация на базе *GROOVE*

В этом разделе описывается, как наша модель реализована в *GROOVE*, используя программный инструмент преобразования графов [9]. Это внедрение затем используется для экспериментирования на примере многоуровневого облачно-ориентированного приложения.

## 4.1. О программном продукте *GROOVE*

*GROOVE* использует простые графы для моделирования структуры объектно-ориентированных систем во время их проектирования, компиляции и выполнения. Графы имеют вершины и дуги, которые могут быть маркированы. Преобразования графов позволяют выразить изменения моделей или для операционной семантики систем. В нашей работе *GROOVE* используется в типизированном режиме, чтобы гарантировать правильное типизирование графов. Этот режим предоставляет общие типы для переменных и правила над графами, которые позволяют управлять присвоенными приоритетами таким образом, что правило данного приоритета применяется, только если никакое правило с более высоким приоритетом не может быть применено к текущему графу.

Графы преобразовываются по правилам, имеющим *a*) шаблоны, которые должны присутствовать (соотв. отсутствовать) для применения правила, *b*) элементы (вершины и дуги), для добавления или удаления в графе, и *c*) пары вершин, которые будут слиты. Кодировка, использующая цвета и формы, позволяет дать более наглядное представление этих правил. Например, примитивная операция *add* моделируется графовым правилом, представленным на рис. 3. На этом рисунке показаны

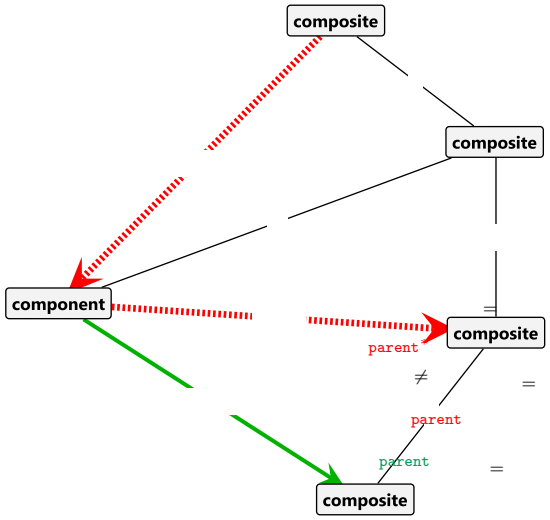
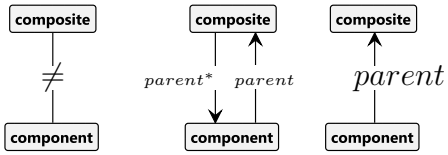



Рис. 3: Примитивное реконфигурирование *add* в *GROOVE*
Fig. 3. *add* primitive operation in *GROOVE*




(a) LHS (b) NAC (c) RHS

Рис. 4: Эквивалент правила *GROOVE* на рис. 3, использующий графы LHS, NAC и RHS
Fig. 4. Equivalent of *GROOVE* rule in Fig. 3 using LHS, NAC, and RHS graphs

*a*) компонент и составной компонент; дуги с маркировкой "=" гарантируют, что вершины для составных компонентов имеют одинаковый тип *composite*, в то время как дуги с маркировкой "≠" гарантируют, что вершины для обычных компонентов имеют тип *component*, отличающийся от типа для составных компонентов; *b*) запрещающие красные штриховые дуги с маркировкой "*parent**" или "*parent*" гарантируют отсутствие (транзитивного) отношения *parent* между вершинами с маркировкой "composite" и "component". Если вышеупомянутые условия удовлетворены, зеленая дуга с маркировкой "*parent*" создается между вершинами "component" и "composite".

Такое правило преобразований графа может быть представлено с помощью грамматического правила, имеющего *a*) LHS подграф (левая часть правила) для представления предварительных условий правила, *b*) NAC подграф (отрицательное условие для применения), определяющий то, что не должно произойти в соответствии с правилом, и *c*) RHS подграф (правая часть правила) для представления выходных условий. Подграфы LHS, NAC и RHS, выражающие правило *GROOVE* на рис. 3, изображены на рис. 4.

Входными данными для нашего внедрения является граф компонентно-ориентированной системы, представленной с использованием модели из раздела 3. Такой граф показывает конфигурацию, как описано в определении 1, где элементы и отношения представляются соответственно вершинами и дугами.

## 4.2. Возращаясь к примеру

Возвратимся к VM, представленной на рис. 1. Она смоделирована составным компонентом $virtualMachine$, который может содержать подкомпоненты, представляющие службы приложения $httpServer$, $appServer$ или $dataServer$. Эта VM может также содержать *наблюдателей*, которые являются подкомпонентами для мониторинга служб. Подкомпонент $osObs$ используется для мониторинга операционной системы VM, и он может быть связан с подкомпонентами $httpObs$, $appObs$ или $dataObs$, используемыми соответственно для мониторинга служб подкомпонентов $httpServer$, $appServer$ и $dataServer$.

Таблица 4: Принцип генерации кода инсталляции

Table 4: Install code generation principle

| **Feature** | **data** | **app** | **http** | **managed** |
|---|---|---|---|---|
| **Bit #** | 3 | 2 | 1 | 0 |
| **ic = 0** | 0 | 0 | 0 | 0 |
| **ic = 5** | 0 | $2^2 = 4$ | 0 | $2^0 = 1$ |
| **ic = 10** | $2^3 = 8$ | 0 | $2^1 = 2$ | 0 |
| **ic = 11** | $2^3 = 8$ | 0 | $2^1 = 2$ | $2^0 = 1$ |

Каждая VM имеет свои особенности (опции), определенные бинарным *кодом инсталляции* $ic$, каждый бит которого действует как флаг для включения или отключения определенной опции. Это представлено в таблице 4, где первая линия показывает опции, а вторая показывает номер соответствующего бита. Последующие

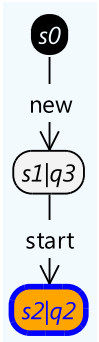



Рис. 5: Пустая ОС
Fig. 5. Bare OS ($ic = 0$)

линии описывают генерацию кода инсталляции для сервера с пустой операционной системой ($ic = 0$), с управляемым сервером приложения ($ic = 5$) и с управляемым (соотв. неуправляемым) LAMP (Linux+Apache+MySQL+PHP) сервером, имеющим код инсталляции 10 (соотв. 11).

Наша программная реализация создает модель компонентно-ориентированной системы, представляющей VM, специфицированную данным кодом инсталляции. Рисунок 5 показывает систему с переходами над графами, полученную с помощью *GROOVE* во время создания VM с пустой ОС ($ic = 0$). Первое состояние (`s0`) представляет пустой граф; `s1` обозначает граф, представляющий составной компонент

VM в остановленном состоянии; s2 представляет граф с тем же самым компонентом VM после запуска. Переходы маркированы выполняемыми примитивными операциями по реконфигурированию.

Точно так же для компонентно-ориентированной системы, представляющей сервер управляемого приложения ($ic = 5$), система с переходами над графами показана на рис. 6. В дополнение к примитивным операциям по реконфигурированию, которые используются для маркировки переходов, вводится этикетка "chk_present_appServerPC" для подтверждения, присутствует ли подкомпонент сервера приложения или нет. Таким образом, используя язык управления *GROOVE*, функция *manage()* добавляет и формирует подкомпоненты с адекватным мониторингом.

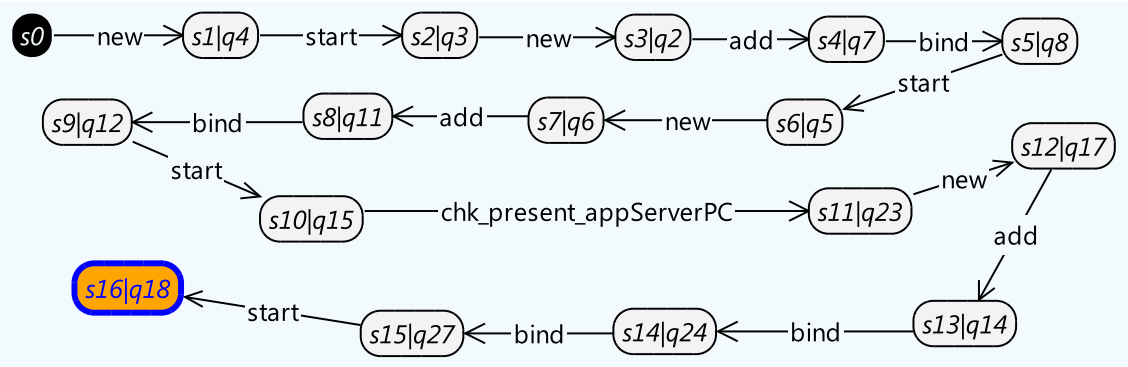



Рис. 6: Сервер управляемого приложения ($ic = 5$)
Fig. 6. Managed application server ($ic = 5$)

Для VM с более чем одним сервисным компонентом, такой как LAMP сервер ($ic = 10$ или $ic = 11$), имеющей http и службу данных, порядок связывания и запуска этих компонентов не является предопределенным, как показано на рис. 7. Изменения сначала происходят детерминированным образом от состояния s0 до s8. Состояние s16, в правом верхнем углу, обозначает граф, соответствующий спецификации для установочного кода 10, т.е. для неуправляемого LAMP сервера. Начиная с этого состояния может быть вызвана *GROOVE* функция *manage()*, чтобы получить управляемый LAMP сервер ($ic = 11$) между s23 и s41. Заметим, что изменения между s8 и s16 недетерминированы. Мы располагаем двумя кратчайшими путями (s8 → s10 → s16 и s8 → s11 → s16), которые могут быть легко обнаружены, используя исследование в ширину.

Для каждого кода инсталляции таблица 5 показывает количество состояний и переходов системы с переходами над графами. У системы с переходами над графами для $ic = 11$, показанной на рис. 7, имеется 26 состояний и 66 переходов. Заметим, что в нашей программной реализации порядок примитивных операций по реконфигурированию полностью определен для $0 \leq ic \leq 5$ и $8 \leq ic \leq 9$.

В таблице 5 для $6 \leq ic \leq 7$ и $10 \leq ic \leq 13$ мы наблюдаем для каждой VM с двумя услугами, что управляемая версия имеет на 16 состояний и переходов больше, чем неуправляемая. Подобный вывод может быть сделан, рассматривая последнюю линию таблицы 5. Это показывает, как изображено на рис. 7, что *GROOVE* функция *manage()* полностью определяет порядок примитивных операций по реконфигурированию. Заметим, что количество состояний и переходов для $ic = 6$ (соотв. $ic = 7$),

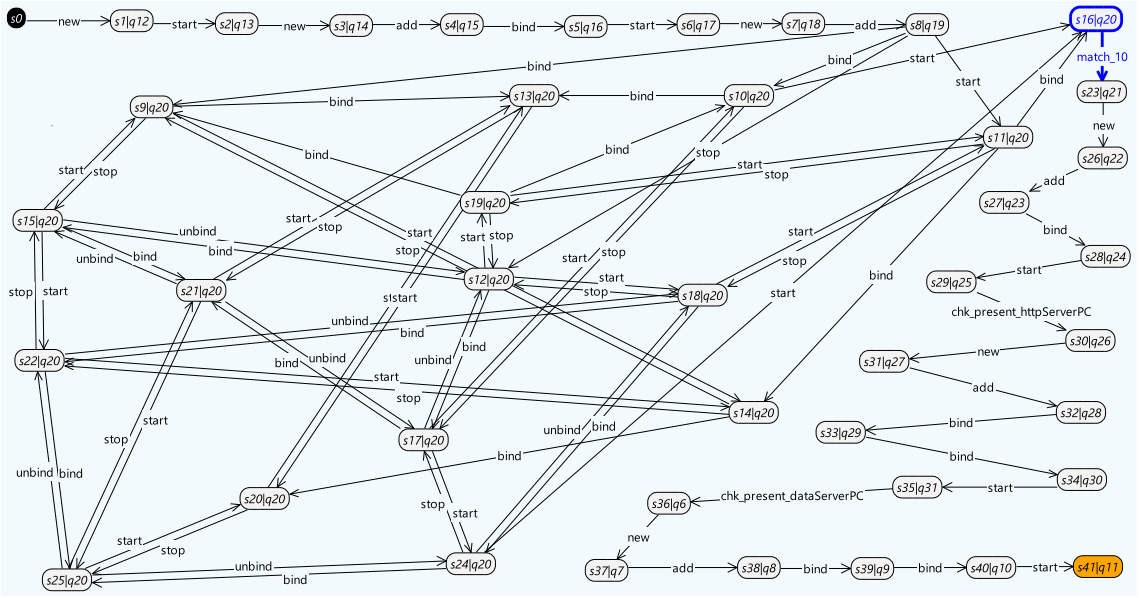



Рис. 7: Управляемый LAMP сервер ($ic = 11$)
Fig. 7. Managed LAMP server ($ic = 11$)

Таблица 5: Количество состояний и переходов для кода инсталляции

Table 5. Number of states and transitions per install code

| **Unmanaged** | | | **Managed** | | |
|---|---|---|---|---|---|
| *ic* | states | transitions | *ic* | states | transitions |
| 0 | 3 | 2 | 1 | 7 | 6 |
| 2 | 7 | 6 | 3 | 17 | 16 |
| 4 | 7 | 6 | 5 | 17 | 16 |
| 6 | 46 | 155 | 7 | 62 | 171 |
| 8 | 7 | 6 | 9 | 17 | 16 |
| 10 | 26 | 66 | 11 | 42 | 82 |
| 12 | 26 | 66 | 13 | 42 | 82 |
| 14 | 265 | 1456 | 15 | 288 | 1479 |

отличается от их количества для $ic = 10$ или $ic = 12$ (соотв. $ic = 11$ или $ic = 13$). Это объясняется тем фактом, что в отличие от *httpServer* или *appServer*, подкомпонент *dataServer* не имеет требуемого интерфейса (см. рис. 1), что приводит к дополнительному детерминизму, для достижения конфигурации, включающей этот компонент.

## 5. Имплементация vs. спецификация

В модели спецификации примитивные операции и охраняемые реконфигурирования оставались достаточно абстрактными, и операция *run* не интерпретировась. Формальная семантика компонентно-ориентированной системы с интерпретируемыми

операциями может быть получена, просто обогащая конфигурации более точной памятью состояний и эффектом этих операций на память.

## 5.1. Интерпретируемые конфигурации и реконфигурирования

Пусть $GM = \{u...\}$ – множество (бесконечное в общем случае) состояний общей глобальной памяти, и $LM = \{v...\}$ – множество (бесконечное в общем случае) состояний локальной памяти данного компонента. Эти состояния памяти читаются и изменяются при выполнении примитивных и непримитивных реконфигурирований, а также операций, конкретизирующих $run$. Формально все действия $ope \in \mathcal{R}_{run}$ интерпретированы как отображения $\overline{ope}$ из $GM \times LM$ на себя. Кроме того, существуют некоторые действия, свойственные имплементации $\mathcal{R}_{imp}$, такие как $manage$ из раздела 4. Назовем $\mathcal{I} = (GM, LM, (\overline{ope})_{ope \in \mathcal{R}_{run} \cup \mathcal{R}_{imp}})$ интерпретацией базового множества $\mathcal{R}_{run}$. Пусть $\mathcal{I}_{\mathcal{R}_{run}} = \{\mathcal{I}, \mathcal{I}_{GROOVE}...\}$ обозначает класс всех интерпретаций, в частности содержащий $\mathcal{I}_{GROOVE}$ для интерпретации $GROOVE$.

*Интерпретируемые конфигурации.* В дополнение к уже интерпретируемым параметрам и интерфейсам (см. [7] для дополнительной информации), состояние компонентов может быть описано более точно при помощи состояний локальной памяти. Множество интерпретируемых состояний компонентов есть наименьшее множество $State_{\mathcal{I}}$ такое что, если $s_1, \ldots, s_n$ – элементы в $State$[6], $v_1, \ldots, v_n \in LM$ – состояния локальной памяти, тогда $((s_1, v_1), \ldots, (s_n, v_n))$ являются состоянием в $State_{\mathcal{I}}$. Затем множество интерпретируемых конфигураций $\mathcal{C}_{\mathcal{I}}$ определяется через $GM \times State_{\mathcal{I}}$.

*Интерпретируемые переходы.* Предположим, что все примитивные действия имеют детерминированный эффект на локальную и глобальную память, всегда заканчиваются (либо нормально, либо с исключением), и дают результат.

Для $\mathcal{I}_{GROOVE}$ из раздела 4, каждый граф представляет интерпретируемую конфигурацию, соответствующую конфигурации, данной в определении 1, тогда как переходы между конфигурациями осуществляются, используя правила над графами.

Для каждой примитивной операции реконфигурирования $ope$ соответствующее правило над графами, обозначенное $\overline{ope}$, имеет эквивалентные или более строгие предварительные условия. Например, для примитивной операции $add$ предварительные условия (3) и (4) из таблицы 2 представлены графами LHS и NAC (рис. 4a и 4b) соответствующего правила над графами, тогда как выходное условие (1) изображено на рис. 4c. Предварительные условия (2) и (5) неявно определены типированием правила, которое содержит вершину типа $component$[7](соотв. $composite$), соответствующие компонентам $comp_1$ (соотв. $comp_2$) в таблице 2. Поскольку обе вершины в правиле над графами наследуют тип $component$, предварительное условие (2) выполнено. Кроме того, тот факт, что вершина, соответствующая $comp_2$, типирована как $composite$, гарантирует, что она не содержит параметров, и таким образом удовлетворяет предварительное условие (5).

[6]Рассматриваемое как отношение.

[7]Так как это абстрактный тип, вершина типа *component* является либо *примитивной*, либо *составной*.

Кроме того, отметим на рис. 4b, в дополнение к дуге с маркировкой $parent^*$, удовлетворяющей предварительное условие (4), другую дугу с маркировкой *parent*, гарантирующей, что вершина типа *composite* не является родителем другой вершины, т.е. $(comp_1, comp_2) \notin Parent$. Это условие не присутствует в таблице 2 из-за спецификации на базе множеств. Однако без этого NAC $(comp_1, comp_2) \notin Parent$ имплементация на базе *GROOVE* может закончить с двумя дугами, маркированными *parent* между вершиной типа *component* и вершиной типа *composite*. Это могло бы произвести граф, не соответствующий спецификации из определения 1.

Наконец, все конструкции теперь ведут себя детерминированно, и недетерминированное глобальное поведение производится произвольным чередованием действий компонентов. Это приводит к следующему определению.

**Определение 4** (Семантика имплементации)**.** *Операционная семантика имплементации определяется системой с переходами* $S_{\mathcal{I}} = \langle \mathcal{C}_{\mathcal{I}}, \mathcal{C}^0_{\mathcal{I}}, \mathcal{R}_{run_{\mathcal{I}}}, \rightarrow_{\mathcal{I}}, l_{\mathcal{I}} \rangle$*, где* $\mathcal{C}_{\mathcal{I}}$ *– множество конфигураций вместе с их состояниями памяти,* $\mathcal{C}^0_{\mathcal{I}}$ *– множество начальных конфигураций,* $\mathcal{R}_{run_{\mathcal{I}}} = \{\overline{ope} \mid ope \in \mathcal{R}_{run} \cup \mathcal{R}_{imp}\}$*,* $\rightarrow_{\mathcal{I}} \subseteq \mathcal{C}_{\mathcal{I}} \times \mathcal{R}_{run_{\mathcal{I}}} \times \mathcal{C}_{\mathcal{I}}$ *– отношение реконфигурирования, соблюдающее правила над графами, и* $l_{\mathcal{I}} : \mathcal{C}_{\mathcal{I}} \rightarrow CP$ *– всюду определенная функция интерпретации.*

## 5.2. Корректные имплементации и сохранение непротиворечивости

Существуют прочные связи между интерпретируемой моделью и моделью спецификации. В этом разделе будет установлено, что имплементация на базе *GROOVE* соответствует спецификации.

Пусть $\mathcal{R}_{run_{\mathcal{I}}} = \{\overline{ope} \mid ope \in \mathcal{R}_{run} \cup \mathcal{R}_{imp}\}$ – множество, в котором $\overline{ope}$ принадлежит конечному множеству охраняемых реконфигурирований, использующих примитивные правила над графами, конкретизированные при имплементации системы. Для имплементации на базе *GROOVE* рассмотрим $\mathcal{R}_{imp} = \{match\}$, где *match* представляет операции для определения значений условий для реконфигурирования, используемых в *GROOVE*. Это не изменяет текущий граф, подобно "chk_present_appServerPC" на рис. 6 в потоке управления.

Чтобы установить связи между интерпретируемой моделью и моделью спецификации, мы предлагаем использовать версию классической $\tau$-симуляции [16], маркируя как $\tau$ операции в $\mathcal{R}_{imp}$. Для всех $ope \in \mathcal{R}$ обозначим $c \overset{ope}{\Rightarrow} c'$, когда есть $n, m \geq 0$, такие что $c \overset{\tau^n ope \tau^m}{\longrightarrow} c'$.

**Определение 5** ($\tau$-симуляция)**.** *Пусть* $S_1$ *и* $S_2$ *– две модели над* $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$*. Бинарное отношение* $\sqsubseteq_\tau \subseteq \mathcal{C}_1 \times \mathcal{C}_2$ *называется* $\tau$*-симуляцией тогда и только тогда, когда для любой операции* $ope$ *в* $\mathcal{R}$*,* $(c_1, c_2) \in \sqsubseteq_\tau$ *влечет: когда* $c_1 \overset{ope}{\Rightarrow}_1 c'_1$*, тогда существует* $c'_2 \in \mathcal{C}_2$*, такое что* $c_2 \overset{ope}{\Rightarrow}_2 c'_2$ *и* $(c'_1, c'_2) \in \sqsubseteq_\tau$*.*

Определим $S_1$ и $S_2$ как $\tau$-подобные, что обозначено $S_1 \sqsubseteq_\tau S_2$, если $\forall c^0_1 \in \mathcal{C}^0_1 \exists c^0_2 \in \mathcal{C}^0_2 . (c^0_1, c^0_2) \in \sqsubseteq_\tau$.

Рассмотрим интерпретируемые операции по реконфигурированию в $\mathcal{R}_{run_{\mathcal{I}}}$ и соответствующие им неинтерпретируемые операции. Маркируя операции в $\mathcal{R}_{imp}$ через $\tau$, мы можем утверждать следующее:

**Теорема 1** (Моделирование)**.** $S_{\mathcal{I}} \sqsubseteq_{\tau} S$.

*Набросок.* Установим сначала, как это сделано выше для операции *add*, что в имплементации у любой примитивной операции по реконфигурированию имеются более сильные предварительные условия, чем у соответствующей операции в модели спецификации. Таким образом, мы можем доказать[8], что охраняемые реконфигурирования, состоящие из примитивных операторов $G \to \overline{s}$, с $\overline{s} \in \mathcal{R}_{run_{\mathcal{I}}} \backslash \mathcal{R}_{imp}$, имеют более строгие предварительные условия, чем соответствующие операторы $s \in \mathcal{R}_{run}$.

Рассмотрим последовательность охраняемых реконфигурирований $G_0 \to \overline{s}_0, G_1 \to \overline{s}_1, \ldots, G_n \to \overline{s}_n$, игнорируя закрывающие действия $\tau$ в $\mathcal{R}_{imp}$. Для всех $i$, таких что $0 \leq i \leq n$, $\overline{s}_i$ имеет более строгие предварительные условия, чем $s_i$; таким образом, существует $G'_i$, т.ч. $G_i \to G'_i$ и $G'_i \to s_i$, как иллюстрировано ниже.

$$\begin{array}{cccccccccc} S_{\mathcal{I}} & G_0 & \to & \overline{s}_0, & G_1 & \to & \overline{s}_1, & \ldots, & G_n & \to & \overline{s}_n \\ & \Downarrow & & & \Downarrow & & & & \Downarrow & & \\ S & G'_0 & \to & s_0, & G'_1 & \to & s_1, & \ldots, & G'_n & \to & s_n \end{array}$$

Поскольку закрывающие действия $\tau$ в $\mathcal{R}_{imp}$ вводятся, чтобы оценить условия последовательностей охраняемых реконфигурирований, они не формируют бесконечных циклов, состоящих только из $\tau$-переходов. Таким образом, всегда существует выход из этих циклов, если таковые имеются, с помощью перехода с этикеткой $\overline{ope}$, и этот переход всегда в конечном счете выполнен, что заканчивает доказательство. □

Этот результат показывает, что модель спецификации является корректным приближением к более реалистичной интерпретируемой модели. Поскольку свойства достижимости совместимы с $\sqsubseteq_{\tau}$, это приводит нас, следовательно, к следующему результату:

**Утверждение 2** (Корректность и полнота)**.**

- *Если конфигурация $c$ не достижима в $S$, то она не достижима ни в какой $S_{\mathcal{I}}$.*
- *С другой стороны, если конфигурация $c$ достижима в $S$, то существует интерпретация $\mathcal{I}$, такая что $c$ достижима в $S_{\mathcal{I}}$.*

Как следствие теоремы 1 и утверждений 1 и 2, верен следующий результат:

**Утверждение 3.** *Пусть $S_{\mathcal{I}} = \langle \mathcal{C}_{\mathcal{I}}, \mathcal{C}^0_{\mathcal{I}}, \mathcal{R}_{run_{\mathcal{I}}}, \to_{\mathcal{I}}, l_{\mathcal{I}} \rangle$ – интерпретируемая модель, и $S = \langle \mathcal{C}, \mathcal{C}^0, \mathcal{R}_{run}, \to, l \rangle$ – модель спецификации. При $\mathcal{C}^0_{\mathcal{I}} \subseteq \mathcal{C}_{\mathcal{I}}$, если $S_{\mathcal{I}} \sqsubseteq_{\tau} S$, тогда* `consistent`$(\mathcal{C}^0_{\mathcal{I}})$ *влечет* `conistent`$(reach(\mathcal{C}^0_{\mathcal{I}}))$.

[8]Используя гипотезу о самых слабых предварительных условиях, как определено в [8].

# 6. Библиографические заметки и заключение

## 6.1. Библиографические заметки

Самоадаптация – важная и активная область исследования с приложениями в различных областях. В работе [1] подчеркнута важная проблема, состоящая в устранении разрыва между разработкой и имплементацией адаптивных систем. Настоящая статья показывает, что платформа *GROOVE* может помочь устранить этот разрыв. В [5] рекофигурирование компонентно-ориентированных систем осуществлялось во времени выполнения, используя политики адаптации, приводимые в действие с помощью темпоральных шаблонов. Однако рассмотренные в этой работе реконфигурирования были просто последовательностями примитивных операций по реконфигурированию. В настоящей работе, где используются альтернативные и повторяющиеся конструкции для составления реконфигурирований, реконфигурирования могут иметь различные результаты. Это зависит от контекста или может быть вызвано недетерминированными механизмами при имплементации. В данной работе это не только статическая последовательность инструкций реконфигурирования (как это имело место в [5, 10, 12, 17]), но действительно *динамическое* реконфигурирование.

В отличие от [18], мы предполагаем, что реконфигурирования не всегда ведут компоновку компонентов от одной непротиворечивой архитектуры к другой. Таким образом, мы рассматриваем более общий случай реконфигурирований.

*Непротиворечивость версий* была введена в [17], чтобы минимизировать прерывание службы (*disruption*) и задержки, с которыми компонентно-ориентированные (распределенные) системы обновляются (*timeliness*) посредством реконфигурирований. Она квалифицирует состояние, где транзакции в пределах системы таковы, что данное реконфигурирование может не разрушить систему и произойти в ограниченное время; непротиворечивость версии была связана с *неподвижностью* [19] и *спокойствием* [20] с намерением собрать лучшее из обоих понятий. В отличие от [17, 19, 20], мы рассматриваем архитектурные ограничения как предварительные условия для применения охраняемых реконфигурирований. Таким образом, рассматривая компоненты как черные ящики, принцип разделения проблем принят во внимание. Аппликативная непротиворечивость, связанная с транзакциями в пределах системы или с внешними событиями, может сохраняться во время выполнения (*runtime*), используя механизмы политик адаптации, как описано в [5] для централизованной системы и [6] для децентрализованных или распределенных систем.

Следуя за [21], наше понятие непротиворечивости может быть рассмотрено как специальный архитектурный стиль. Тем не менее, при использовании графовых грамматик мы представляем типы интерфейсов компонентно-ориентированных систем специальными вершинами графов. Таким образом, подобно [22] мы можем осуществлять мониторинг темпоральных свойств на уровне интерфейсов.

## 6.2. Заключение

Следуя за [8], в данной работе предложена грамматика для охраняемых реконфигурирований. Она позволяет создавать реконфигурирования на базе примитивных операций по реконфигурированию, используя последовательности реконфигурирований, а также альтернативные и повторяющиеся конструкции. Определение самых слабых предварительных условий для применения реконфигурирований позволило нам доказать, что эти охраняемые реконфигурирования сохраняют непротиворечивость конфигураций.

В данной работе также представлена имплементация модели, используя программный инструмент преобразования графов *GROOVE* [9], где компонентно-ориентированные системы представляются как графы, элементы (например, компоненты, интерфейс, параметры, и т.д.) представлены вершинами, а отношения между элементами (например, $Parent$, $Bindings$, и т.д.) представлены дугами. Эта имплементация, используемая для работы с примером управляемой/неуправляемой облачной среды, позволяет моделировать реконфигурирование правилами над графами, используя LHS, RHS и NAC подграфы. Когда различные результаты могут произойти для каждого реконфигурирования, множество возможных выполнений может быть представлено как граф LTS, используя нашу имплементацию под *GROOVE*; возможные состояния, т.е. конфигурации системы под наблюдением, показаны как вершины, а реконфигурирования между ними как переходы.

По отношению к модели реконфигурирования, корректность интерпретируемых систем также была доказана с использованием правил грамматик над графами для выполнения реконфигурирования, что демонстрирует корректность нашей имплементации под *GROOVE*.

Возможное направление дальнейшего исследования включает анализ вышеупомянутых графов LTS, чтобы обнаружить или предотвратить формирование циклов в рамках реконфигурирований. Мы также планируем реализовать динамические реконфигурирования, основанные на грамматиках графа для приложения политик адаптации во время выполнения компонентно-ориентированных систем.

# A. Доказательство утверждения 1 для конструкций на базе примитивных утверждений

Рассмотрим множество $grs$ охраняемых реконфигурирований, использующих охраняемые списки, составленные только из примитивных утверждений. Это множество $grs$ содержит $B'_1 \to S_1 [\!] \dots [\!] B'_n \to S_n$ с булевым $B'_i$ и $Si = ope^i_0; ope^i_1; \dots; ope^i_{n_i}$, где $n_i$ представляет количество примитивных утверждений $(ope^i_0, ope^i_1, \dots, ope^i_{n_i})$ охраняемого списка $S_i$, и $R^i_j$ (соотв. $R^i_{j+1}$) представляет предварительное условие (соотв. выходное условие) $ope^i_j$ для $0 < i \leq n$ и $0 \leq j \leq n_i$ (соотв. $0 < j \leq n_i + 1$).

Так как $R^i_0$ – предварительное условие $S_i$, и конфигурация перед применением $S_i$ рассматривается как непротиворечивая, $grs$ переписывается в виде $B_1 \to S_1 [\!] \dots [\!] B_n \to S_n$, с $B_i = B'_i \wedge CC \wedge R^i_0$. Определим $BB = (\exists i : 1 \leq i \leq n : Bi)$, а также множества $I = \{i \in \mathbb{N}.1 \leq i \leq n\}$ и $I_\top = \{i \in I.Bi\}$.

## Альтернативная конструкция

Пусть $IF$ обозначает **if** $B_1 \to S_1 [\!] \ldots [\!] B_n \to S_n$ **fi**. По определению, $wp(IF, R) = BB \wedge \forall i \in I : B_i \Rightarrow wp(S_i, R)$. Ранее было установлено, что для последовательности примитивных утверждений $S_i$ верно $CC \wedge R_0^i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i)$; тогда по определению $B_i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i) \Rightarrow wp(S_i, CC)$. Это означает, что $wp(IF, CC) = BB \wedge \forall i \in I : B_i \Rightarrow wp(S_i, CC)$, и этого достаточно, чтобы доказать, что непротиворечивость сохраняется альтернативной конструкцией.

Однако для альтернативной конструкции можно установить более строгое выходное условие $CC \wedge \bigwedge\limits_{i \in I_\top} R_{n_i+1}^i$, полагая, что каждый член конъюнкции $\bigwedge\limits_{i \in I_\top} R_{n_i+1}^i$ является частью условия охраняемого списка, имеющего право на выполнение.

Тогда $wp(IF, CC \wedge \bigwedge\limits_{j \in I_\top} R_{n_j+1}^j) = BB \wedge \forall i \in I : B_i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i)$, так как по определению $\forall i \in I_\top, B_i = \top$.

Следовательно:

$$\begin{aligned} BB \wedge CC \wedge \bigwedge_{i \in I_\top} R_0^i \;&\Rightarrow\; BB \wedge \bigwedge_{i \in I_\top} wp(S_i, CC \wedge R_{n_i+1}^i) \\ &\Rightarrow\; BB \wedge (\forall i \in I : B_i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i)) \\ &\Rightarrow\; wp(IF, CC \wedge \bigwedge_{j \in I_\top} R_{n_j+1}^j) \end{aligned}$$

В качестве примера возьмем частный случай альтернативной конструкции **if** $B$ **then** $S$ **fi**, записанный в виде **if** $B \to S [\!] \neg B \to skip$ **fi**. Его самое слабое предварительное условие – $wp(\textbf{if } B \textbf{ then } S \textbf{ fi}, CC \wedge (B \Rightarrow post_S)) = B \wedge wp(S, CC \wedge R_S)$, где $R_S$ и $post_S$ – соответственно предварительное условие и выходное условие для $S$.

## Повторяющаяся конструкция

Пусть $DO$ обозначает **do** $B_1 \to S_1 [\!] \ldots [\!] B_n \to S_n$ **od**. Пусть $H_0(R) = R \wedge \neg B$, и для $k > 0$, пусть $H_k(R) = wp(IF, H_{k-1}(R)) \vee H_0(R)$, где $IF$ обозначает ту же самую охраняемую конфигурацию, заключенную между "**if fi**". Тогда по определению $wp(DO, R) = \exists k : k > 0 : H_k(R)$.

Это означает, что самое слабое предварительное условие этой конструкции гарантирует надлежащее завершение после как максимум $k$ выборов охраняемого списка, оставляя систему в состоянии, удовлетворяющем $R$. Рассмотрим $l_0, l_1, \ldots, l_k$, такие что для $0 \le j \le k$, $1 \le l_j \le n$ и $S_{l_0}; S_{l_1}; \ldots; S_{l_k}$ в качестве упорядоченной последовательности утверждений, выбранных при выполнении конструкции до ее завершения. Перед этим мы доказали, что такая последовательность сохраняет непротиворечивость. Поэтому $CC \wedge R_{n_{l_k}+1}^{l_k}$ является истинным выходным условием, и так как с $CC \wedge R_{n_{l_k}+1}^{l_k} \Rightarrow CC \wedge \bigvee\limits_{i \in I} R_{n_i+1}^i$, мы имеем $wp(DO, CC \wedge R_{n_{l_k}+1}^{l_k}) \Rightarrow wp(DO, CC \wedge \bigvee\limits_{i \in I} R_{n_i+1}^i)$.

Для последовательности примитивных заявлений $S_i$ мы установили перед этим, что $CC \wedge R_0^i \Rightarrow wp(S_i, CC \wedge R_{n_i+1}^i)$; тогда $CC \wedge R_0^{l_0} \Rightarrow wp(S_{l_0}; S_{l_1}; \ldots; S_{l_k}, CC \wedge R_{n_{l_k}+1}^{l_k})$.

Следовательно:

$$\begin{aligned} CC \wedge \bigwedge_{i\in I} R_0^i &\Rightarrow CC \wedge R_0^{l_0} \\ &\Rightarrow wp(S_{l_0}; S_{l_1}; \ldots; S_{l_k}, CC \wedge R^{l_k}_{n_{l_k}+1}) \\ &\qquad\qquad \text{for any valid sequence } S_{l_0}; S_{l_1}; \ldots; S_{l_k} \\ &\Rightarrow wp(DO, CC \wedge R^{l_k}_{n_{l_k}+1}) \\ &\Rightarrow wp(DO, CC \wedge \bigvee_{i\in I} R^i_{n_i+1}) \end{aligned}$$

Это доказывает, что повторяющаяся конструкция, примененная к множеству охраняемых реконфигурирований, использующих списки примитивных утверждений, сохраняет непротиворечивость. Это также обеспечивает более строгие предварительные и выходные условия, которые используются для завершения доказательства утверждения 1.

# Список литературы

[1] De Lemos R., Giese H., Müller H. A., Shaw M., Andersson J., Litoiu M., Schmerl B., Tamura G., Villegas N. M., Vogel T., et al., “Software engineering for self-adaptive systems: A second research roadmap”, *Software Engineering for Self-Adaptive Systems II*, 2013, 1–32.

[2] Dwyer M. B., Avrunin G. S., Corbett J. C., “Patterns in property specifications for finite-state verification”, Proceedings of the 1999 International Conference on, *Software Engineering*, 1999, 411–420.

[3] Dormoy J., Kouchnarenko O., Lanoix A., “Runtime verification of temporal patterns for dynamic reconfigurations of components”, LNCS, **7253**, eds. Arbab F., Ölveczky P., Springer, Berlin Heidelberg, 2012, 115–132.

[4] Trentelman K., Huisman M., “Extending JML specifications with temporal logic”, *Algebraic Methodology and Software Technology*, 2002, 334–348.

[5] Kouchnarenko O., Weber J. F., “Adapting component-based systems at runtime via policies with temporal patterns”, LNCS, **8348**, eds. Fiadeiro J. L., Liu Z., Xue J., Springer, 2014, 234–253.

[6] Kouchnarenko O., Weber J. F., “Decentralised evaluation of temporal patterns over component-based systems at runtime”, LNCS, **8997**, eds. Lanese I., Madelaine E., Springer, 2015, 108–126.

[7] Lanoix A., Dormoy J., Kouchnarenko O., “Combining proof and model-checking to validate reconfigurable architectures”, *ENTCS*, **279** (2011), 43–57.

[8] Dijkstra E. W., “Guarded commands, nondeterminacy and formal derivation of programs”, *Communications of the ACM*, **18** (1975), 453–457.

[9] Ghamarian A. H., de Mol M., Rensink A., Zambon E., Zimakova M., “Modelling and analysis using GROOVE”, *J. on Software Tools for Technology Transfer*, **14** (2012), 15–40.

[10] Bruneton E., Coupaye T., Leclercq M., Quéma V., Stefani J. B., “The fractal component model and its support in java”, *Softw., Pract. Exper.*, **36** (2006), 1257–1284.

[11] Schneider S., Treharne H., “Csp theorems for communicating b machines”, *Formal Asp. Comput.*, **17** (2005), 390–422.

[12] Seinturier L., Merle P., Rouvoy R., Romero D., Schiavoni V., Stefani J.B., “A component-based middleware platform for reconfigurable service-oriented architectures”, *Software: Practice and Experience*, **42** (2012), 559–583.

[13] Dormoy J., Kouchnarenko O., Lanoix A., "Using temporal logic for dynamic reconfigurations of components", LNCS, **6921**, eds. Barbosa L., Lumpe M., Springer, Berlin Heidelberg, 2012, 200–217.

[14] Hamilton A. G., *Logic for mathematicians*, Cambridge University Press, Cambridge, 1978.

[15] Hoare C. A. R., "An axiomatic basis for computer programming", *Communications of the ACM*, **12** (1969), 576–580.

[16] Milner R., *Communication and concurrency*, Prentice-Hall, Inc., 1989.

[17] Ma X., Baresi L., Ghezzi C., Panzica La Manna V., Lu J., "Version-consistent dynamic reconfiguration of component-based distributed systems", *ACM*, Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, 2011, 245–255.

[18] Boyer F., Gruber O., Pous D., "Robust reconfigurations of component assemblies", Int. Conf. on Software Engineering (ICSE '13, Piscataway), 2013, 13–22.

[19] Kramer J., Magee J., "The evolving philosophers problem: Dynamic change management", *Software Engineering, IEEE Transactions*, **16** (1990), 1293–1306.

[20] Vandewoude Y., Ebraert P., Berbers Y., D'Hondt T., "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates", *Software Engineering, IEEE Transactions*, **33** (2007), 856–868.

[21] Le Metayer D., "Describing software architecture styles using graph grammars", *IEEE Transactions on Software Engineering*, **24** (1998), 521–533.

[22] Kähkönen K., Lampinen J., Heljanko K., Niemelä I., "The lime interface specification language and runtime monitoring tool", *Int. Wshop on Runtime Verification, RV'09*, LNCS, **5779**, eds. Bensalem S., Peled D.A., Springer, Berlin Heidelberg, 2009, 93–100.

---

**Kouchnarenko O., Weber J.-F.**, "Component-based Systems Reconfigurations Using Graph Grammars", *Modeling and Analysis of Information Systems*, **23**:6 (2016), 804–825.

**DOI:** 10.18255/1818-1015-2016-6-804-825

**Abstract.** Dynamic reconfigurations can modify the architecture of component-based systems without incurring any system downtime. In this context, the main contribution of the present article is the establishment of correctness results proving component-based systems reconfigurations using graph grammars. New guarded reconfigurations allow us to build reconfigurations based on primitive reconfiguration operations using sequences of reconfigurations and the alternative and the repetitive constructs, while preserving configuration consistency. A practical contribution consists of the implementation of a component-based model using the *GROOVE* graph transformation tool. Then, after enriching the model with interpreted configurations and reconfigurations in a consistency compatible manner, a simulation relation is exploited to validate component systems' implementations. This sound implementation is illustrated on a cloud-based multi-tier application hosting environment managed as a component-based system.

**Keywords:** component-based systems, dynamic reconfigurations, consistency, simulation relation, implementation, *GROOVE*

**About the authors:**
Olga Kouchnarenko, orcid.org/0000-0003-1482-9015, PhD,
Universite de Bourgogne - Franche-Comte,
16 route de Gray, 25000 Besancon, France, e-mail: olga.kouchnarenko@univ-fcomte.fr

Jean-Francois Weber, graduate student,
Universite de Bourgogne - Franche-Comte,
16 route de Gray, 25000 Besancon,France, e-mail: jfweber@femto-st.fr

**Acknowledgments:**
This work has been partially funded by the Labex ACTION, ANR-11-LABEX-0001-01.