УДК 004.413

Формальный логический анализ корректности спецификаций сетевых SIP-протоколов

© В.В. Девятков, Т.Н. Мьё

МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

В статье для проверки правильности и корректности описания SIP-спецификаций (Session Initiation Protocol) в отличие от известных работ предлагается использовать значительно более выразительный, хорошо структурированный и как формальная система более развитой вариант языка, основанный на моделях взаимодействующих последовательностных процессов (π-исчислений). Спецификации должны удовлетворять определенным свойствам, которые описываются на языке временной модальной логики. Поиск ошибок предлагается осуществлять не с помощью генерации трасс, а с помощью доказательства наличия указанных формальных свойств. Ошибкой предлагается считать отсутствие таких свойств. Процессные модели позволяют гораздо более четко и полно классифицировать и описывать типы ошибок. В качестве инструментария для поиска ошибок предлагается использовать язык логического программирования ПРОЛОГ, что является гораздо более изящным и не имеющим ограничений подходом к проверке правильности и корректности спецификаций.

Ключевые слова: *последовательностный процесс, пользовательский агент, язык временной модальной логики, протокол инициирования сеанса, язык логического программирования ПРОЛОГ.*

Введение. В последние годы протокол SIP (Session Initiation Protocol) стал широко применяться для IP-мультимедиауслуг, включая обмен мультимедийным содержимым (видео- и аудиоконференции, мгновенные сообщения, онлайн-игры). Стандарт SIP описан во многих источниках [1, 2]. В модели взаимодействия открытых систем SIP является сетевым протоколом прикладного уровня. Однако документация, имеющаяся по протоколу, плохо структурирована, неформальна и неполна, что затрудняет ее использование как средства проверки корректности конкретных приложений SIP.

Для выхода из этой ситуации необходимо формализовать описание функционирования приложений с помощью соответствующих моделей, позволяющих проводить формально проверку корректности (правильности) функционирования. Имеется достаточно много работ, посвященных решению этой проблемы.

Так, в работе [3] для формальной спецификации протоколов TCP, UDP и их приложений предлагается использовать *язык логики предикатов высоких порядков*, порождая на основе этого языка необходи-

мые для того или иного протокола соответствующие формальные системы и методы доказательства корректности. Недостатки такого подхода широко известны, в частности, из работ по искусственному интеллекту: это трудноразрешимость или даже неразрешимость, вычислительная сложность, необходимость соответствующей квалификации пользователя, сложность описания функционирования приложений и др.

Практически во всех описаниях протокола SIP его функционирование представляют как множество последовательностных параллельно взаимодействующих процессов (агентов). При таком описании как пользовательский агент *UAC* на стороне клиента (user agent on the client side), так и пользовательский агент *UAS* на стороне сервера (user agent on the server side) моделируются своим последовательностным процессом.

В ряде других работ предлагаются более простые модели описания взаимодействующих агентов. Так, в [4] для этой цели предлагается использовать язык Promela. Авторы указанной работы полагают, что этот язык удобен для спецификации SIP-приложений по двум причинам. Во-первых, выразительные возможности языка в наибольшей степени соответствуют описанию функционирования как множества последовательностных параллельно взаимодействующих агентов, принятому в SIP. Во-вторых, описание на языке Promela можно проверить на правильность с помощью специальных программ проверки Spin [5].

В настоящей статье для проверки SIP-спецификаций также предлагается использовать модель последовательностных взаимодействующих агентов, однако для описания спецификаций в отличие от работы [4] предлагается применять значительно более выразительный, хорошо структурированный и теоретически, как формальная система, более развитой вариант языка, основанный на моделях взаимодействующих последовательностных процессов (π-исчислении [6]). Спецификации должны удовлетворять определенным свойствам, описанным на языке временной модальной логики. Поиск ошибок предлагается осуществлять не с помощью генерации трасс, как это делается в работе [4], а с помощью доказательства наличия описанных формальных свойств [6, 7]. Ошибкой предлагается считать отсутствие таких свойств. Процессные модели в дальнейшем будем называть просто процессами. Процессы позволяют гораздо более четко и полно классифицировать и описывать типы ошибок. В качестве инструментария для поиска ошибок будем использовать язык логического программирования ПРОЛОГ, что является гораздо более изящным и не имеющим ограничений подходом к проверке правильности и корректности спецификаций.

Настоящая статья организована следующим образом. В разделе 1 дается краткое описание модели последовательностных взаимодействующих процессов, используемой для SIP-спецификаций. Раздел 2 посвящен принципам процессного описания спецификаций. Раздел 3 представляет принципы перехода от процессного описания спецификаций к описанию на языке ПРОЛОГ и поиск ошибок в SIP-спецификациях по описанию на этом языке.

1. Процессные модели. Каждый процесс имеет алфавит восприятий и реакций $A = \{a_1, a_2, ..., a_m\}$. Каждый символ a этого алфавита именует некоторый объект, либо получаемый (воспринимаемый) процессом из внешней среды (восприятие процесса), либо выдаваемый им во внешнюю среду (внешняя реакция процесса), либо используемый для внутренних нужд (внутренняя реакция процесса). Процессы действуют, воспринимая, порождая для внутреннего употребления или выдавая наружу объекты с соответствующими именами. Для того чтобы различать типы действий, будем использовать следующие обозначения: $?a$ — для восприятий; $!a$ — для внешних реакций; b — для внутренних реакций.

Нитью a^* будем называть кортеж (конечный или бесконечный) действий $a^* = a_0a_1a_2...a_{m-2}a_{m-1}$. *Выполнением нити процессом* P называется определенная последовательность выполнения ее действий слева направо. Символом e обозначается *пустое действие*. Нить, состоящая из единственного пустого действия, называется *пустой нитью. Процессом* P называется множество нитей S, которые он может выполнять, а *поведением процесса* P — порядок выполнения множества этих нитей.

Введем следующие обозначения:

$?A$ — множество всех восприятий некоторого процесса P, включая пустое восприятие $?e$;

$!A$ — конечное множество всех внешних реакций процесса P, включая пустую внешнюю реакцию $!e$;

S — множество всех нитей, выполняемых процессом P, таких что $S \subseteq ?A^* \times !A$, где $?A^*$ — множество всех нитей $?a^*$ в алфавите $?A$ ($?a^*$ — нить, состоящая только из восприятий (нить восприятий) процесса P);

φ^* — функция на множестве $?A^*$, которая ставит в соответствие каждой нити $?a^* \in ?A^*$ внешнюю реакцию из множества $!A$.

Процесс P, выполняющий множество нитей $?a^*\varphi^*(?a^*) \in S$, будем обозначать $P(S)$. Значение функции φ^* на тех нитях множества $?A^*$, на которых эта функция не определена, и на тех нитях, кото-

рые этому множеству не принадлежат, считается равным $!e$. Множество нитей $?a^*$ таких, что $\{?a^* \in ?A^* \mid ?a^*\varphi^*(?a^*) \in S\}$, будем обозначать $?S$.

Популярным языком представления процессов является *язык графов переходов*. Для построения графа переходов процесса считается, что после каждого его восприятия, в том числе пустого, процесс осуществляет внутреннюю реакцию или, как говорят, переходит во внутреннее состояние ожидания следующего восприятия. Находясь в этом внутреннем состоянии, процесс может порождать внешнюю реакцию, в том числе пустую. В гра́фе переходов каждое состояние процесса изображается кружком, внутрь которого помещается символ этого состояния; каждому восприятию соответствует стрелка, соединяющая состояния этого перехода. Начальное состояние выделяется двойным кружком. На рис. 1 показан граф переходов некоторого процесса P.

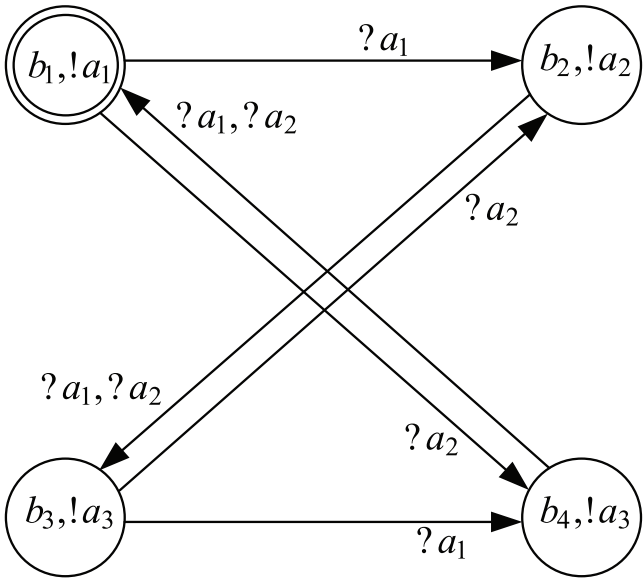

Рис. 1. Граф переходов процесса P

Граф переходов процесса позволяет компактно описывать небольшое множество нитей, в том числе бесконечных. В этих условиях естественным кажется описывать процесс непосредственно его графом переходов. Однако при большой размерности графа такое описание становится громоздким и ненаглядным. Альтернативой этому служит использование адекватных графу канонических процессных выражений, позволяющих легко переходить от графа к этим выражениям и наоборот.

Так, для графа переходов, приведенного на рис. 1, каноническими процессными выражениями, адекватными этому графу, будут следующие:

$$P \square \begin{pmatrix} b_1 = ?e \\ b_1 = b_4 ? a_1 \\ b_1 = b_4 ? a_2 \\ b_2 = b_1 ? a_1 \\ b_2 = b_3 ? a_2 \\ b_3 = b_2 ? a_1 \\ b_3 = b_2 ? a_2 \\ b_4 = b_1 ? a_2 \\ b_4 = b_3 ? a_1 \\ !a_1 = b_1 \\ !a_2 = b_2 \\ !a_3 = b_3 \,|\, b_4 \end{pmatrix},$$

где каждое внутреннее процессное выражение вида $b_i = b_j ? a_k$ задает один переход из состояния b_j в состояние b_i в результате восприятия $?a_k$. Выражение $!a_i = b_j$ задает реакцию $!a_i$ после перехода процесса в состояние b_j.

Если исходным описанием процесса являются канонические процессные выражения, то переход от этих выражений к графу переходов процесса и наоборот очевиден. Канонические процессные выражения могут быть рекурсивными.

2. Принципы процессного описания спецификаций. В настоящей статье, как и в работе [2], модели могут содержать множество пользовательских агентов на стороне клиентов *UAC* и множество пользовательских агентов на стороне серверов *UAS*. Взаимодействие агентов осуществляется с помощью обмена сообщениями. Каждый агент является последовательностным процессом, параллельно функционирующим с другими.

Как уже говорилось во введении, главной задачей моделирования является автоматизация разработки правильных и корректных приложений на основе использования языков, обеспечивающих высокий уровень абстрагирования описания поведения от ненужных деталей, например от деталей на уровне стеков протоколов.

При этом модель считаем *правильной*, если взаимодействие агентов удовлетворяет определенным условиям, а под *корректностью* подразумеваем удовлетворение каждого агента некоторым формальным спецификациям, описывающим его индивидуальное поведение.

В терминах нитей модель будет *правильной*, если каждый агент в процессе взаимодействия с другими агентами выполняет только те

нити, которые разработчик считает допустимыми, и *корректной*, если все эти нити одного агента удовлетворяют некоторым формальным спецификациям. В рамках настоящей статьи будем полагать, что как свойства правильности, так и свойства корректности удастся формально специфицировать и, следовательно, формально проверить.

Пример простой процессной модели. Для иллюстрации описания поведения простой процессной модели воспользуемся парой агентов *UAC* и *UAS*, взятой из работы [2], и сначала опишем их поведение на языке графов, а затем на языке процессных выражений. Агенты являются параллельно выполняющимися последовательностными процессами *UAC* и *UAS*, образующими в совокупности процесс $P \triangleq UAC \| UAS$. Процессы *UAC* и *UAS* взаимодействуют по 10 каналам, имена которых *ackc, brpc, irpc, reqc, sakc, acks, brps, irps, reqs, saks*. Каналы *ackc, brpc, irpc, reqc, sakc* являются входными для процесса *UAC* и выходными для процесса *UAS*, а каналы *acks, brps, irps, reqs, saks,* наоборот, — выходными для процесса *UAC* и входными для процесса *UAS*. Полагаем, что каждый из процессов может выполнять только шесть типов действий: *invite*, *invSucc* (любой из 2*nn* ответов на действие *invite*), *invFail* (любой из 3*nn*—6*nn* ответов на действие *invite*), *ack*, *bye*, *byeRsp* (любой из 200 OK ответов на действие *buy*).

Если какое-либо действие *a* берется из канала с именем *c*, то оно является *восприятием* и записывается как *c*?*a*, если же оно выдается в канал *c*, то оно является *внешней реакцией* и записывается как *c*!*a*. Если действие *a* является внутренней реакцией, то оно записывается просто как !*a*. Имена процессов начинаются с прописной буквы.

Граф переходов процесса *UAC* представлен на рис. 2, а процесса *UAS* — на рис. 3. Процессные выражения *UAC* и *UAS* приведены соответственно на рис. 4 и 5.

Рассмотрим сначала принцип работы последовательностного процесса *UAC* (см. рис. 2 и 4). Вследствие большей компактности принцип работы процесса будем пояснять, используя процессные выражения рис. 4. Граф переходов, изображенный на рис. 2, позволяет наглядно следить за работой процесса.

До начала работы процесса *UAC* ни один из его последовательностных подпроцессов не выполняется. Только один из подпроцессов может выполняться в дальнейшем в любой текущий момент времени. Поэтому будем говорить, что когда выполняется какой-либо подпроцесс, например *Inviting*, то процесс *UAC* находится в состоянии *Inviting*.

После запуска на выполнение процесса *UAC* начинает выполняться нить *UAC.reqc!invite* подпроцесса *Inviting*, в результате чего в канал *reqc* выдается действие (внешняя реакция) *!invite*, нить *UAC.reqc!invite* закончит свое выполнение и начинает выполняться

процесс *Inviting* (подпроцесс процесса *UAC*). Это означает, что могут выполняться следующие нити:

- *Inviting.brps*?*byeRsp*!*assertF* процесса *Inviting*;
- *Inviting.irps*?*invSucc.ackc*!*ack* процесса *Confirming*;
- *Inviting.irps*?*invFail*, *Inviting.reqs*?*bye.brpc*!*byeRsp* процесса *PreEnding*.

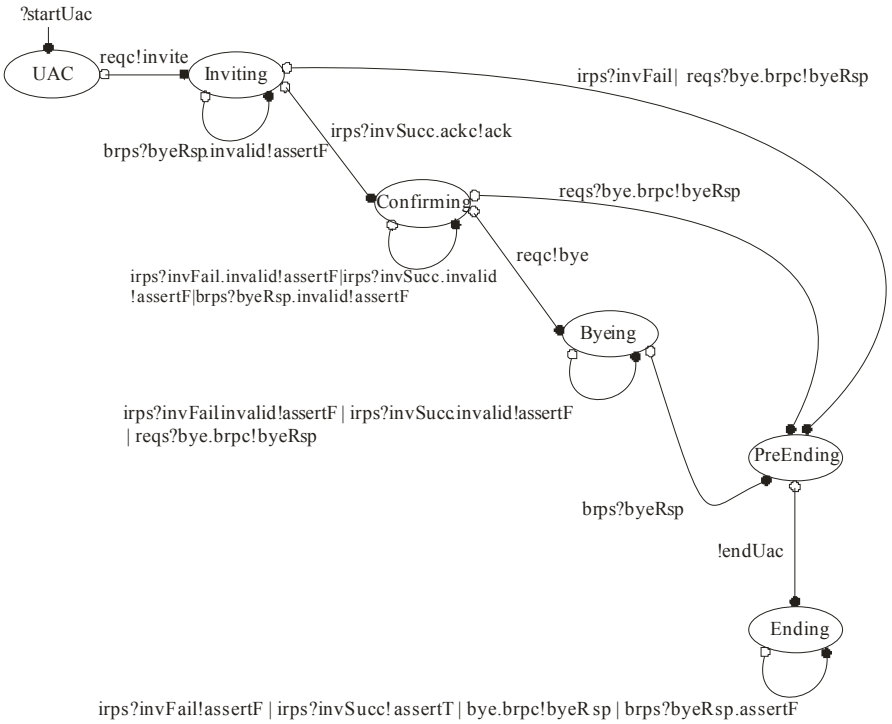

Рис. 2. Граф переходов процесса *UAC*

Проверка на выполнимость осуществляется в порядке записи нитей в процессных выражениях слева направо. Если будет выполнена нить процесса *Inviting*, то процесс проверки названных нитей начинается сначала. Если будет выполнена какая-либо из нитей другого процесса, то процесс *Inviting* прекращает свое выполнение и начинает выполняться процесс, которому принадлежит выполнившаяся нить, и все повторяется сначала, но уже для другого процесса.

Графы процессов не задают порядок выполнения нитей процессов. Проверку выполнимости нитей в случае описания агентов графами переходов можно осуществлять либо в любом порядке, помня при этом, что поведение агентов может быть неоднозначным, либо в порядке, все-таки как-то заданном, например, за счет ортогональности выполнения нитей. В рамках настоящей статьи будем полагать, что имеющиеся процессные выражения всегда задают порядок выполнения нитей.

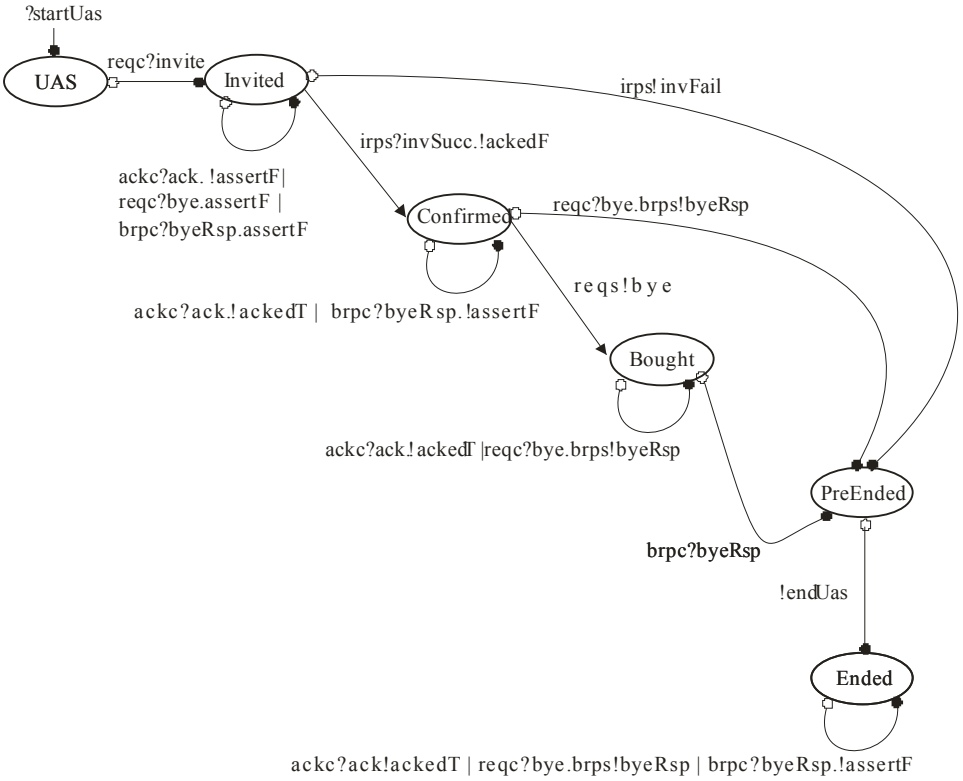

Рис. 3. Граф переходов процесса *UAS*

UAC □ *start?startUac,*

Inviting □ *UAC.reqc!invite*
|*Inviting.brps?byeRsp.invalid!assertF*,

Confirming □ *Inviting.irps?invSucc.ackc!ack*
|*Confirming.irps?invFail.invalid!assertF*|*Confirming.irps?invSucc.invalid!assertF*
|*Confirming.brps?byeRsp.inavalid!assertF*,

Byeing □ *Confirming.reqc!bye*
|*Byeing.irps?invFail.invalid!asseretF*|*Byeing.irps?invSucc.invalid!assertF*
|*Byeing.reqs?bye.brpc!byeRsp*

PreEnding □ *Inviting.irps?invFail*|*Inviting.reqs?bye.brpc!byeRsp*
|*Comfirming.reqs?bye.brpc!byeRsp*
|*Byeing.brps?byeRsp,*

Ending □ *PreEnding!endUac*
|*Ending.irps?invFail!assertT*|*Ending.irps?invSucc!assertT*
|*Ending.reqs?bye.brpc!byeRsp*|*Ending.brps?byeRsp!assertF*.

Рис. 4. Процессные выражения *UAC*

$UAS \Box startUas,$

$Invited \Box UAS.reqc!invite$
$\quad |Invited.ackc?ack.!assertF|Invited.reqc?bye.assertF|Invited.brpc?byeRsp.assertF,$

$Confirmed \Box Invited.irps?invSucc.!ackedF$
$\quad |Confirmed.ackc?ack!ackedT|Confirmed.brpc?byeRsp!assertF,$

$Bought \Box Confirmed.reqs!bye$
$\quad |Bought.ackc?ack!ackedT|Bought.reqc?bye.brps!byeRsp,$

$PreEnded \Box Invited.irps!invFail$
$\quad |Confirmed.reqc?bye.brps!byeRsp$
$\quad |Bought.brpc?byeRsp,$

$Ended \Box PreEnded.!endUas$
$\quad |Ended.ackc?ack.!assertT|Ended.reqc?bye.brps!byeRsp|Ended.brpc?byeRsp!assertF.$

Рис. 5. Процессные выражения *UAS*

3. Принципы перехода от процессного описания спецификаций к описанию на языке ПРОЛОГ и поиск некорректностей. Переход от процессного описания SIP-спецификаций к описанию на языке ПРОЛОГ включает следующие основные этапы:

• построение по процессному описанию агентов (спецификаций) логической программы на языке ПРОЛОГ (от начального раздела до раздела GOAL);

• построение по описанию на языке модальной логики раздела GOAL;

• проверка свойства.

3.1. ***Построение по процессному описанию агентов (SIP-спецификаций) логической программы на языке ПРОЛОГ.*** Принципы перехода от процессного описания к программе на языке ПРОЛОГ рассмотрим для одного процесса (агента), например для процесса *uac* (см. рис. 4). Он начинает выполняться после выполнения нити *start?startuac.* Поэтому в разделе *clauses* логической программы вводим правило

uac([put(start,startuac)]):-thread([put(start,startuac)]),

означающее, что при истинности предиката *thread*([*put*(*start,startuac*)]), аргументом которого является составной объект *put*(*start,startuac*), соответствующий нити *start?startuac*, становится истинным предикат *uac*([*put*(*start,startuac*)]), что интерпретируется как начало выполне-

ния процесса *uac*. Аналогично для процесса *inviting* (см. рис. 4) вводим два правила:

inviting([put(reqc,invite)]):
uac([put(start,startuac)]), thread([put(reqc,invite)])

и

inviting([put(brps,byersp),put(invalid,assertf)]):
inviting([put(reqc,invite)]),thread([put(brps,byersp),put(invalid,assertf)]).

Число этих правил совпадет с числом нитей, после выполнения которых и при условии, что агент находится в состоянии выполнения процесса *uac*, начинает выполняться процесс *inviting*. Аналогично вводятся правила для всех остальных процессов (*confirming*, *byeing, preending, ending*). В дополнение к этому вводим факты с предикатным символом *thread*, соответствующие всем нитям на рис. 4. В разделах *domains* и *predicates* описываем все используемые в разделе *clauses* типы аргументов и предикатов.

```
domains
channel=symbol
message=symbol
put=put(channel,message)
list =put*

predicates
nondeterm uac(list)
nondeterm inviting(list)
nondeterm confirming(list)
nondeterm byeing(list)
nondeterm preending(list)
nondeterm ending(list)
nondeterm thread(list)

clauses
thread([put(start,startuac)]).
thread([put(reqc,invite)]).
thread([put(irps,invfail)]).
thread([put(reqs,bye),put(brpc,byersp)]).
thread([put(brps,byersp),put(invalid,assertf)]).
thread([put(irps,incsuc),put(ackc,ack)]).
thread([put(irps,invfail),put(invalid,assertf)]).
thread([put(irps,invsucc),put(invalid,assertf)]).
thread([put(brps,byersp),put(invalid,assertf)]).
```

```
thread([put(reqs,bye),put(brpc,byersp)]).
thread([put(irps,invsucc), put(ackc,ack)]).
thread([put(reqc,bye)]).
thread([put(reqs,bye),put(brpc,byersp)]).
thread([put(brps,byersp)]).
thread([put(end,enduac)]).

uac([put(start,startuac)]):-thread([put(start,startuac)]).

inviting([put(reqc,invite)]):-
uac([put(start,startuac)]),thread([put(reqc,invite)]).
inviting([put(brps,byersp),put(invalid,assertf)]):-
inviting([put(reqc,invite)]),thread([put(brps,byersp),put(invalid,assertf)]).

confirming([put(irps,incsuc),put(ackc,ack)]):-
inviting([put(reqc,invite)]),thread([put(irps,incsuc),put(ackc,ack)]).
confirming([put(irps,incsuc),put(ackc,ack)]):-
inviting([put(brps,byersp),put(invalid,assertf)]),thread([put(irps,incsuc),
put(ackc,ack)]).

confirming([put(irps,invfail),put(invalid,assertf)]):-
confirming([put(irps,invfail),put(invalid,assertf)]),thread([put(irps,invfail),
put(invalid,assertf)]).
confirming([put(irps,invsucc),put(invalid,assertf)]):-
confirming([put(irps,invfail),put(invalid,assertf)]),thread([put(irps,invsucc),
put(invalid,assertf)]).
confirming([put(brps,byersp),put(invalid,assertf)]):-
confirming([put(irps,invfail),put(invalid,assertf)]),thread([put(brps,byersp),
put(invalid,assertf)]).

confirming([put(irps,invfail),put(invalid,assertf)]):-
confirming([put(irps,invsucc),put(invalid,assertf)]),thread([put(irps,invfail),
put(invalid,assertf)]).
confirming([put(irps,invsucc),put(invalid,assertf)]):-
confirming([put(irps,invsucc),put(invalid,assertf)]),thread([put(irps,invsucc),
put(invalid,assertf)]).
confirming([put(brps,byersp),put(invalid,assertf)]):-
confirming([put(irps,invsucc),put(invalid,assertf)]),thread([put(brps,byersp),
put(invalid,assertf)]).

confirming([put(irps,invfail),put(invalid,assertf)]):-
confirming([put(brps,byersp),put(invalid,assertf)]),thread([put(irps,invfail),
put(invalid,assertf)]).
```

```
confirming([put(irps,invsucc),put(invalid,assertf)]):-
confirming([put(brps,byersp),put(invalid,assertf)]),thread([put(irps,invsucc),
put(invalid,assertf)]).
confirming([put(brps,byersp),put(invalid,assertf)]):-
confirming([put(brps,byersp),put(invalid,assertf)]),thread([put(brps,byersp),
put(invalid,assertf)]).

byeing([put(reqc,bye)]):-
confirming([put(irps,incsucc),put(ackc,ack)]),thread([put(reqc,bye)]).
byeing([put(reqc,bye)]):-
confirming([put(irps,invfail),put(invalid,assertf)]),thread([put(reqc,bye)]).
byeing([put(reqc,bye)]):-
confirming([put(irps,invsucc),put(invalid,assertf)]),thread([put(reqc,bye)]).
byeing([put(reqc,bye)]):-
confirming([put(brps,byersp),put(invalid,assertf)]),thread([put(reqc,bye)]).

byeing([put(irps,invifail),put(invalid,assertf)]):-
byeing([put(irps,invinvalid),put(invalid,assertf)]),thread([put(irps,invinvalid),
put(invalid,assertf)]).
byeing([put(irps,invsucc),put(invalid,assertf)]):-
byeing([put(irps,invinvalid),put(invalid,assertf)]),thread([put(irps,invsucc),
put(invalid,assertf)]).
byeing([put(reqs,bye),put(brpc,byersp)]):-
byeing([put(irps,invinvalid),put(invalid,assertf)]),
thread([put(reqs,bye),put(brpc,byersp)]).

byeing([put(irps,invfail),put(invalid,assertf)]):-
byeing([put(irps,invsucc),put(invalid,assertf)]),thread([put(irps,invinvalid),
put(invalid,assertf)]).
byeing([put(irps,invsucc),put(invalid,assertf)]):-
byeing([put(irps,invsucc),put(invalid,assertf)]),thread([put(irps,invsucc),
put(invalid,assertf)]).
byeing([put(reqs,bye),put(brpc,byersp)]):-
byeing([put(irps,invsucc),put(invalid,assertf)]),
thread([put(reqs,bye),put(brpc,byersp)]).

byeing([put(irps,invfail),put(invalid,assertf)]):-
byeing([put(reqs,bye),put(brpc,byersp)]),thread([put(irps,invinvalid),
put(invalid,assertf)]).
byeing([put(irps,invsucc),put(invalid,assertf)]):-
byeing([put(reqs,bye),put(brpc,byersp)]),thread([put(irps,invsucc),
put(invalid,assertf)]).
byeing([put(reqs,bye),put(brpc,byersp)]):-
```

```
byeing([put(reqs,bye),put(brpc,byersp)]),
thread([put(reqs,bye),put(brpc,byersp)]).

preending([put(reqs,bye),put(brpc,byersp)]):-
confirming([put(irps,incsuc),put(ackc,ack)]),thread([put(reqs,bye),
put(brpc,byersp)]).
preending([put(reqs,bye),put(brpc,byersp)]):-
confirming([put(irps,invfail),put(invalid,assertf)]),thread([put(reqs,bye),
put(brpc,byersp)]).
preending([put(reqs,bye),put(brpc,byersp)]):-
confirming([put(irps,invfail),put(invalid,assertf)]),thread([put(reqs,bye),
put(brpc,byersp)]).
preending([put(reqs,bye),put(brpc,byersp)]):-
confirming([put(brps,byersp),put(invalid,assertf)]),thread([put(reqs,bye),
put(brpc,byersp)]).

preending([put(irps,invfail)]):-
inviting([put(reqc,invite)]),thread([put(irps,invfail)]).
preending([put(irps,invfail)]):-
inviting([put(brps,byersp),put(invalid,assertf)]),thread([put(irps,invfail)]).

preending([put(reqs,bye),put(brpc,byersp)]):-
inviting([put(reqc,invite)]),thread([put(reqs,bye),put(brpc,byersp)]).
preending([put(reqs,bye),put(brpc,byersp)]):-
inviting([put(brps,byersp),put(invalid,assertf)]),thread([put(reqs,bye),
put(brpc,byersp)]).

preending([put(brps,byersp)]):-
byeing([put(reqc,bye)]),thread([put(brps,byersp)]).
preending([put(brps,byersp)]):-
byeing([put(irps,invifail),put(invalid,assertf)]),thread([put(brps,byersp)]).
preending([put(brps,byersp)]):-
byeing([put(irps,invsucc),put(invalid,assertf)]),thread([put(brps,byersp)]).
preending([put(brps,byersp)]):-
byeing([put(reqs,bye),put(brpc,byersp)]),thread([put(brps,byersp)]).

ending([put(end,enduac)]):-
preending([put(reqs,bye),put(brpc,byersp)]),thread([put(end,enduac)]).
ending([put(end,enduac)]):-
preending([put(irps,invfail)]),thread([put(end,enduac)]).
ending([put(end,enduac)]):-
preending([put(reqs,bye),put(brpc,byersp)]),thread([put(end,enduac)]).
```

ending([put(end,enduac)]):-
preending([put(brps,byersp)]),thread([put(end,enduac)]).

3.2. ***Построение по описанию на языке модальной логики раздела GOAL.*** В рамках настоящей работы не ставилось цели описания на языке модальной логики всех свойств или требований, которым должны удовлетворять процессные SIP-спецификации. Они могут быть самыми различными, кроме того, они достаточно подробно перечислены в литературных источниках [8]. Принципы использования модальной логики для описания указанных свойств также хорошо известны [9]. Например, если агент *UAC*, начав процесс *uac* имеет хотя бы одну нить, при выполнении которой он обязательно переходит к процессу *inviting*, то это свойство можно выразить простой модальной формулой □◊*inviting*(*X*), где *inviting*(*X*) — предикат, который становится истинным при выполнении процесса *inviting* в результате выполнения нити *X*. Это означает, что должен существовать хотя бы один путь на графе, показанном на рис. 2, ведущий из вершины *uac* в вершину *inviting*. На языке ПРОЛОГ это можно выразить в виде следующего раздела *goal* (цели):

goal
uac([put(X1,Y1)]),thread([put(X2,Y2)]),inviting([put(X2,Y2)]).

3.3. ***Проверка свойства.*** Полученная по процессному описанию логическая программа на языке ПРОЛОГ при заданной цели

uac([put(X1,Y1)]), thread([put(X2,Y2)]), inviting([put(X2,Y2)])

выдает следующий результат:

X1 = start, Y1 = startuac, X2 = reqc, Y2 = invite,

означающий, что существует нить *start*(*startuac*).*reqc*(*invite*), выполнив которую процесс *UAC* переходит после завершения процесса *uac* к выполнению процесса *inviting*.

Заключение. В настоящей статье рассматривается проверка правильности и корректности (ошибок) SIP-спецификаций модели последовательностных взаимодействующих агентов. В отличие от других работ мы используем хорошо структурированный и более развитой вариант языка, основанный на моделях взаимодействующих последовательностных процессов (π-исчислении), язык временно́й модальной логики для описания требований к спецификациям и язык логического программирования ПРОЛОГ для проверки правильности спецификаций. В статье изложены принципы перехода от процессных моделей SIP-спецификаций к логической программе и от требо-

ваний на языке модальной логики к формулировке цели на языке ПРОЛОГ. Приведен пример логической программы и результаты ее работы. В дальнейшем предполагается создать библиотеку требований правильности спецификаций на языке модальной логики и автоматизированную систему проверки правильности SIP-спецификаций, позволяющую непосредственно по процессным моделям и описанию требований на языке модальной логики автоматически переходить к логическим программам и осуществлять проверки правильности.

ЛИТЕРАТУРА

[1] Rosenberg J., Schulzrinne H., Camarillo G., Johnston A. *Session Initiation Protocol (SIP).* IETF Network Working Group Request. for Comments 3261, 2002.

[2] Rosenberg J., Schulzrinne H. *Reliability of provisional responses in Session Initiation Protocol (SIP).* IETF Network Working Group Request. for Comments 3262, 2002.

[3] Bishop S., Fairbairn M., Norrish M., Sewell P., Smith M., Wansbrough K. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP and sockets. *Proc. SIGCOMM'05. ACM*, 2005, August.

[4] Zave P. *Understanding SIP Through Model-Checking*. *Proc. of the 2nd International Conference of Principles, Systems and Applications of IP Telecommunications*. Springer-Verlag, 2008, vol. 5310, pp. 256–279.

[5] Holzmann G.J. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley, 2004, 596 p.

[6] Девятков В.В., Сидякин И.М. Мультиагентная система анализа телеметрической информации. *Вестник МГТУ им. Н.Э. Баумана*, *Сер. Приборостроение,* 2005, № 4 (61), с. 56–85.

[7] Девятков В.В. Построение, оптимизация и модификация процессов. *Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение*, 2012, № 4, с. 60–79.

[8] Chellas B.F. *Modal Logic an Introduction.* The Press Syndicate of the University of Cambridge, 1980, 295 p.

[9] Gabbay D., Hodkinson I., Reynolds M. *Temporal Logic: mathematical foundations and computational aspects*, vol. 1. Clarendon Press, Oxford, 1994.

Статья поступила в редакцию 28.06.2013

Ссылку на эту статью просим оформлять следующим образом:

Девятков В.В., Мьё Т.Н. Формальный логический анализ корректности спецификаций сетевых SIP-протоколов. *Инженерный журнал: наука и инновации*, 2013, вып. 11. URL: http://engjournal.ru/catalog/it/network/999.html

Девятков Владимир Валентинович окончил Ленинградский государственный институт точной механики и оптики в 1963 г. Д-р техн. наук, профессор, заведующий кафедрой «Информационные системы и телекоммуникации» МГТУ им. Н.Э. Баумана. Область научных интересов: системы искусственного интеллекта, мультиагентные системы, распознавание образов. e-mail: deviatkov@iu3. bmstu.ru

Мьё Тхет Наунг родился в 1985 г., окончил МГУ им. М.В. Ломоносова в 2010 г., аспирант второго года кафедры «Информационные системы и телекоммуникации» МГТУ им. Н.Э. Баумана. e-mail: komyothetnaung@gmail.com