Verifying integrity of exception handling in service-oriented software

Tong Xue and Shi Ying
State Key Laboratory of Software Engineering,
Wuhan University,
Wuhan, China
Email: xuetong@whu.edu.cn
Email: yingshi@whu.edu.cn

Qing Wu*
Department of Education Science,
Wuhan University,
Wuhan, China
Email: 12capricom@163.com
*Corresponding author

Xiangyang Jia
State Key Laboratory of Software Engineering,
Wuhan University,
Wuhan, China
and
Guangxi Key Laboratory of Trusted Software,
Guilin University of Electronic Technology,
Guilin, China
Email: jxy@whu.edu.cn

Xiaohui Hu
State Key Laboratory of Software Engineering,
Wuhan University,
Wuhan, China
Email: huxiaohui@whu.edu.cn

Xiaoying Zhai
State Key Laboratory of Software Engineering,
Wuhan University,
Wuhan, China
Email: xiaoyinger18forever@gmail.com

Tao Zhang
Business Support Centre,
Hubei Mobile,
Wuhan, China
Email: cwhzhangtao@163.com

Abstract: In service-oriented software environments, that exceptions may not be fully handled is one of the main causes for system breakdown. Therefore, we need to verify integrity of software exception handling. At present verifying integrity of software exception handling mainly depends upon experiences of developer. Most of automatic formal verification mechanisms can only support some general features, such as equivalence, boundedness, security, etc. and easily cause state space explosion. This paper proposes an integrity verification method of exception handling in service-oriented software. We construct state spaces associated with exception handling and
convert the issue of integrity verification into a model of boundedness analysis based on CPN, and reduce the size of state spaces by extending Stubborn Set and Transition Dependency Graph. An example and experimental results based on extended CPN tools confirm that our method has good generalisation abilities.

**Keywords:** service-oriented software; exception handling; coloured Petri nets; stubborn set; state space; transition dependency graph.


**Biographical notes:** Tong Xue is a PhD candidate with State Key Laboratory of Software Engineering, Wuhan University, China. His research interests include software verification and software architecture.

Shi Ying is a Professor with State Key Laboratory of Software Engineering, Wuhan University, China. He received his PhD degree in Computer Software and Theory from Wuhan University. His research interests include software engineering, cloud computing and dependable software.

Qing Wu is an Associate Professor with Department of Education Science, Wuhan University, China. She received his PhD degree in Computer Software and Theory from Wuhan University. Her research interests include software verification and service computing.

Xiangyang Jia is a Lecturer with State Key Laboratory of Software Engineering, Wuhan University, China. He received his PhD degree in Computer Software and Theory from Wuhan University. His research interests include program analysis, model checking, symbolic execution and service computing.

Xiaohui Hu is a postgraduate student with State Key Laboratory of Software Engineering, Wuhan University, China. Her research interests include model checking and symbolic execution.

Xiaoying Zhai is an undergraduate student with State Key Laboratory of Software Engineering, Wuhan University, China. Her research interests include software testing and verification.

Tao Zhang is an engineer with Business Support Center, Hubei Mobile, China. Her research interests include service computing and software architecture.

1 **Introduction**

Robust exception handling mechanism exerts an important impact on software reliability. In order to ensure all explicit exceptions are handled, it is necessary to construct a formal exception handling model and use appropriate verification technology to assist designers to detect the flaws of exception handling model. Similar requirements are more obviously shown in service-oriented software because of dynamics and uncertainties. The exceptions being thrown out because of defects in exception handling mechanism, propagate and transform layer upon layer, will lead to increased new exceptions with added complexity or even unusual combination of exceptions. If those exceptions are fully handled, we consider this exception handling mechanism has characteristic of integrity. But, under this condition, those exceptions we mentioned above may not be fully handled and that is one of the main causes for system breakdown. Therefore, we need effective method to verify the integrity of exception handling, in order to ensure all exceptions that are defined explicitly are addressed in service-oriented software.

Academia has put forward some formal descriptions and verification methods for exception handling. These methods described and verified exceptions handling logic with various perspectives, including service perspective, process perspective and architecture perspective. Exceptions in service-oriented software can be divided into system exception, resource exception and application exception. The formal verification tools they used included: B method (Castor Filho et al., 2009), CSP (Pereira and de Melo, 2010), LOTOS (Dumez et al., 2013), process algebra (Khazhar and Jalili, 2012), Petri Nets (Jensen, 2011; Kristensen et al., 2004; Kristensen et al., 1998), etc. The formal verification techniques they used are divided into two major categories of manual theorem-proving and automatic state space analysis. Manual theorem-proving is applied to mathematical concept description system. A major restriction of automatic state space analysis is state space explosion (Groote et al., 2015). Owing to large-scale and complexity of state space system analysis model, researchers have also proposed a variety of advanced state space methods to avoid state space explosion, including subsets of state space method and compression of state space method, details will be elaborated in the fourth section in this paper. Meanwhile,
service-oriented software development languages and related specifications have paid a lot of attention to exception handling logic modelling and verification in industry. Such as exception message specifications in WSDL (W3C, 2007b) and SOAP (W3C, 2007a), FaultHandler in BPEL (OASIS, 2007), WS-Coordination (OASIS, 2009), WS-AtomicTransaction (OASIS, 2006a) and WS-BusinessActivity (OASIS, 2006b), etc. These facilities can effectively support programming and testing activity of service-oriented software exception handling.

However, previous efforts have not provided adequate support for exception handling formal integrity verification for service-oriented software. The integrity verification of exception handling mechanisms can mainly be done by developers themselves. Developers verify integrity of software exception handling mainly depends upon writing formulas manually, which is time-consuming and laborious. Manual verification is not applied to prove system error information. Most of automatic formal verification methods can only support some general features, such as boundedness, equivalence, security, etc. A simple use of state space traversal for exception handling verification easily leads to state space explosion. The use of the advanced state space methods depends on characteristics of the system model. None of universal advanced state space methods is applicable to a variety of systems.

Therefore, this paper proposes a method to verify the integrity of service-oriented software exception handling based on coloured Petri nets state space theory. Coloured Petri Nets (CPNs) is a language for the modelling and validation of systems in which concurrency, communication, and synchronisation play a major role. It is possible to verify properties by means of state space methods and model checking (Jensen et al., 2007). CPNs have intuitive graphical modelling capability, and state space analysis methods and validation tools provided by Petri nets are sophisticated. Using CPN tools, user interaction with CPN tools is based on direct manipulation of the graphical representation of the CPN model using interaction techniques, such as tool palettes and marking menus (Jensen et al., 2007).

We convert integrity verification problems into boundedness analysis for a set of specific places and realise automatic verification of exception handling integrity by extending CPN modelling and analysis tools. We put forward a state space construction method based on stubborn set (Valmari, 1991) and dependency graph between transitions (Kristensen and Valmari, 1998) to reduce complexity of the state space analysis. This method extends state space theory of Petri nets stubborn set to the field of CPNs, constructs state spaces that contain exception generating, exception handling and exception throwing according to the exception handling model characteristics of service-oriented software, and reduces complexity of state space analysis by reducing the size of state spaces. An example and experimental results based on extended CPN tools confirm that our method has good generalisation abilities.

Section 2 introduces the definition of service-oriented software exception handling model and its integrity. We propose methods that support to verify the integrity in Section 3. Section 4 gives an example and an experiment to demonstrate exceptions handling logic integrity verification method. Section 5 introduces related work. Finally, this paper gives a summary and future prospect.

2 Service-oriented software exception handling model and integrity

2.1 Service-oriented software exception handling model

Service-oriented software consists of a series of services and processes. These processes provide interactive and collaborative environment and achieve the orchestration of services. Therefore, service-oriented software has obvious hierarchical features. Service-oriented software exception handling includes exception generating, exception handling, and exception re-throwing after failure or returning after correction. Exception in service-oriented software can be generated at different levels in the normal business process, such as Web service or certain part of BPEL process (SCOPE). After the completion of exception handling, service-oriented software flow control will return to the normal business process. Therefore, the formal description of the normal business process provides the foundation for the formal description of exception handling. Owing to the special nature of service-oriented software, service-oriented software exception handling and traditional software exception handling have big differences: (1) from the exception position angle, service-oriented software exceptions may arise from the service layer or certain scope of the process layer, while the traditional software exceptions occur in functions; (2) from the exception handling action angle, compared with traditional software, service-oriented software can take retry or service replacement approach; (3) from the exception handling returning angle, after a service exception in service-oriented software is successfully handled, new message will be delivered directly to the service caller, while the control flow will return to the upper scope in process exception. After the successful handling of internal exception in the function of traditional software, the control flow will return to the calling function; (4) from the exception propagation angle, after service-oriented software service exception handling fails, new exception message will be directly returned to caller services. In the case of process exception handling, the message will be thrown to the upper scope until it is successfully handled or thrown out. When traditional software exception handling fails, a new exception message will be thrown to the caller of the function.

The formal description of service-oriented software exception handling is a hierarchical CPN model. Hierarchical Petri net model consists of formal description of normal business process and formal description of exception handling process. We will give a formal description of the normal business process from two aspects: service and process (Qing,
(1) Service can be seen as a black box. We give formal description of service interactive interface, rather than focusing on the service implementations. Service interface consists of types, message and service portType, while service portType consists of a series of operations. The type elements and message elements are expressed as color sets in hierarchical CPNs. The service operations are expressed as hierarchical Petri net modules. portTypes are expressed as sets of hierarchical Petri net modules of service operations. Therefore, service interfaces are expressed as sets of hierarchical Petri net modules of service portTypes.

(2) The scopes of WS-BPEL process are expressed as hierarchical Petri net modules. From the perspective of the control flow, execute states of the scope are represented by the state places, which will convert the expression of blocking and continuing operations in scope into transitions. From the perspective of the data flow, we translate variable information of scopes into variable places and translate reading and writing data operations in the scope into transitions. We translate external interfaces of scopes into port places of hierarchical Petri net modules and translate the internal interfaces of scopes into socket places of hierarchical Petri net modules. We also translate the interactions between scopes into port-socket relation of different hierarchical Petri net modules. As to exception handling process, the thrown positions described in Web services or BPEL process are expressed as the thrown point places of hierarchical Petri net modules, that is, all exception messages that the modules throw will be thrown in the places. Exception handling process is described as an alternative transition. The transitions can catch all exception messages of thrown point place. As we have mentioned above, the exception handling of Web services and BPEL process are slightly different. (1) For web services, regardless of whether service exceptions have been successfully handled, returned message tokens will return to the service call place. (2) For the scopes of BPEL process, if normal messages return, tokens return to the end state place of hierarchical Petri net modules. If exception messages return, exception message tokens will spread to the upper-level Petri net modules by the port-socket relation of hierarchical Petri net between modules.

Definition 2.1: (Exception handling model in service-oriented software) A hierarchical Coloured Petri Nets model CPNEH=(S, SM, PS, FS) (Wu, 2012), in which:

(1) S is a finite set of modules; each module is a hierarchical Coloured Petri Nets module \( s = ((P^t, T^t, A^t, \Sigma^t, V^t, C^t, G^t, E^t, I^t), T^t_{sub}, P_{sub}, PT^t) \) . For each \( s_1, s_2 \in S \) makes \( s_1 \neq s_2 \) and \( (P^t \cup T^t) \cap (P^t \cup T^t) = \emptyset \) . And for any \( s \in S \) satisfies following conditions:

\[ \begin{align*}
Ps &= \text{businessPlaces} \cup \text{EHPplaces}, \text{businessPlaces}\{\text{responses}, \text{requests}\} \cup \{(\text{starts}, \text{ends}) \cup \text{varPlaces}\}, \text{in which, requests} \in P \text{ stands for request message places in service portTypes, responses} \in P \text{ stands for response message places in service portTypes, starts} \in P \text{ stands for start state places in BPEL process scopes, ends} \in P \text{ stands for end state place in BPEL process scopes, varPlaces} \text{ stand for variable places sets in BPEL process. EHPplaces} = \{\text{ThrowPoints, reThrowPoints}\}, \text{in which, ThrowPoints} \in P \text{ stands for exception thrown point places of module } s, \text{ reThrowPoints} \text{ stands for exception re-throw places of module } s.
\end{align*} \]

\( Ts = \text{businessTransitions} \cup \text{EHTransitions} \), in which \( \text{EHTransitions} \in T^t_{sub} \text{ stands for exception handling action.} \)

(2) \( SM : T^t_{sub} \rightarrow S \) refers to the sub-module functions. For each Web service or BPEL scope, sub-modules are used, module level is acyclic.

(3) \( PS \) is a port-socket association. Each alternative transition \( t \) has a port-socket association function \( PS(t) \subseteq P^t_{sub}(t) \times P^t_{port}(t) \) . For all \( (p, p') \in PS(t), t \in T^t_{sub} \), there are \( ST(p) = PT(p'), ST(p) = PT(p'), C(p) = C(p'), I(p) = I(p') \).

(4) \( FS \subseteq 2^P \) is the non-empty set of fusion set. For all \( p, p' \in fs, fs \in FS \) makes \( C(p) = C(p'), I(p) = I(p') \).

2.2 Integrity of exception handling

Definition 2.2: (Exception handling integrity) all exceptions that declare explicitly are addressed. In exception handling model, if exception tokens generated by services and processes have corresponding exception handling services, and have not been thrown to the final exception thrown places in model, the exception handling is integrated. Exception handling integrity verification transforms into checking the upper bound of token numbers in exception thrown places for zero values. If the upper bound is zero, it indicates that all exceptions have received appropriate treatment in the system eventually, exception handling is integrated.

Because exception handling model is a hierarchical CPN, as port places of modules, exception thrown point places and exception occurred point places in upper modules have port–socket relationship. Therefore, exception thrown point places stand for compound places that are logically equivalent. The final exception thrown point places can be explained as exception messages stored in the compound places will not be consumed by the other transitions, it represents that an exception thrown point compound place does not have subsequent transitions. If exception handling model state space is limited, i.e. nodes and arcs in state spaces are limited. By means of all nodes traversal in state spaces, if the number of tokens in compound places which represent exception thrown points equals zero, then the exception handling is integrated.

Theorem 2.1: exception handling model is a hierarchical Coloured Petri Nets model CPNEH=(S, SM, PS, FS)

\[ \forall \text{ThrowPoint} \in \text{Set} : \forall \text{ThrowPoint}_p \in \text{ThrowPointSet} , \text{meets} \]

\[ \text{ThrowPoint}_p \subseteq P \land (\text{ThrowPoint}_p) = \{0\} \] and \( \forall p \in \text{ThrowPoint}_p \), \( C(p) \) belongs to the exception message type, making \( \forall M \in \mathcal{M}(M_0) : \sum_{p \text{=ThrowPoint}_p} | M(p) | = 0 \), then the exception handling is integrated.
We use proof by contradiction as follows:

If \( \exists p \in \text{ThrowPoint}_{op} \), then \( M(p) > 0 \), exception handling in model is integrated.

\[ \therefore p \in \text{ThrowPoint}_{op} \]

\[ \therefore p \in P \text{ and } C(p) \text{ belongs to exception message type.} \]

\[ \therefore \text{There is at least a Token element } (p, c) \text{, in which } c \in C(p) \text{ is exception message.} \]

\[ \therefore p' = \{ \emptyset \} \]

\[ \therefore \text{Token element } (p, c) \text{ does not have corresponding handling, and is contradictory to definitions 2.2.} \]

3 Stubbenn set state space method for integrity verification

As we have mentioned above, we can use existing tools and methods about Coloured Petri Nets to verify integrity of exception handling by traversing all reachable states in model and calculating the upper bounds of tokens in sets of specific places.

However, state space explosion is a major restriction of state space analysis, especially in large-scale complex systems. We extend previous effort about Petri nets state space stubborn set and propose construction method of state space stubborn set based on TDG. The construction method as follows:

**PHASE 1:** Construct the description of exception handling process based on Coloured Petri Nets model as TDG, which is used to support stubborn sets of boundedness analysis of particular place sets.

**PHASE 2:** Propose stubborn set construction algorithm, treat TDG and current executing states in model as input, and treat the stubborn sets consisting of binding elements sets as output. For each state space node, only the binding elements enabled in stubborn sets can be used to generate post-nodes of state spaces and then construct the state spaces that support integrity verification.

**PHASE 3:** With the help of sophisticated methods of places boundedness analysis, we can verify the integrity of exception handling.

From the above, on one hand, the generation of stubborn sets based on TDG constructs the state spaces that support integrity verification of exception handling, which reduces the number of identities. On the other hand, we use complete traverse of TDG’s nodes and edges as the standard of state space construction, which reduces the size of state spaces. Therefore, the complexity of state space analysis is obviously reduced.

3.1 Work basis

In this section, we refer to the concept of colour mapping proposed by Sami Evangelista (Evangelista and Pradat-Peyre, 2006), and lay the theoretical foundation for the later section in this paper.

**Definition 3.1:** (Colour domain) Basic type \( \Delta \) is limited and non-empty finite set. When \( C_i \in \Delta \) and \( s(C) \in \mathbb{N} \) are the size of \( C \), the colour domain \( C \) is a product of \( C_1 \times \ldots \times C_{|C|} \). We use \( \varepsilon \) to represent colour domain sets.

For example, suppose \( t \in T \), \( Var(t) = \{ x, ok \} \), where Type(x) = \{ index d with 1..4 \} , Type(ok) = BOOL and there is a binding \( b(t) = \{ x = 2, ok = true \} \), then \( \Delta_1 = \{ 1, 2, 3, 4 \} \), \( \Delta_2 = \{ true, false \} \), \( s(C) = 2, C_1 \in \Delta_1 \), \( C_2 \in \Delta_2 \). Then, the colour domain of \( b(t) \) is \( C \), \( c \in C \) \( c = \{ x = 2, ok = true \} \) or \( c = \{ x = 2, true \} \).

Colour mapping of nets is constructed by element expressions. We allow two types of elements expressions: variables and function expression. Variables refer to the selected specific elements in entries of a colour domain. Function expressions provide enabling complex operations of basic types. From the colour domain to basic types expressing can be placed in the element expressions. Variable is a special type of function expression.

**Definition 3.2:** (Element expression) Assume \( C \in \varepsilon \) and \( \delta \in \Delta \), \( e_{c, \delta} \) is the sets of expressions from \( C \) to \( \delta \), \( e_{c, \delta} = V_{c, \delta} \cup F_{c, \delta} \).

\( V_{c, \delta} = \{ V_i | i \in [1..s(C)] \} \cup C_\delta = \delta \) is a set of variables from \( C \) to \( \delta \). \( V_i \) is defined as \( \forall c \in C \), \( V_i(c) = e_i \).

\( F_{c, \delta} = \{ (f, (e_1, \ldots, e_n)) | f \in \delta \times \ldots \times \delta_n \rightarrow \delta \land \forall i \in [1..n], e_i \in e_{c, \delta} \} \) is a function expressions set from \( C \) to \( \delta \). \( f, (e_1, \ldots, e_n) \) is defined as \( \forall c \in C, \forall i \in [1..n], e_i \in e_{c, \delta} \) and \( f(e_1, \ldots, e_n)(c) = f(e_1(c), \ldots, e_n(c)) \).

**Definition 3.3:** (Element expression tuple) element expression tuple is a triple tuple \( tup = (\gamma, \alpha, E) \), of which \( \gamma \in G_T \) is a guard function of the tuple, \( \alpha \in \mathbb{N}^+ \) is coefficient value of the element expressions, \( E \) is element expression. If \( \forall c, c' \in C \) and \( tup(c) = c' \), then \( E \equiv e_{c, \delta} \).

If \( i \in [1..s(C')] \), element expression \( e_i \in e_{c, \delta} \), \( tup(c) = \{ (\gamma, \alpha, < e_{i_1}, \ldots, e_{i_{n'}}) \} \).

Tuples \( (\gamma, \alpha, < e_1, \ldots, e_n) \) can be expressed as \( \chi \cdot \alpha < e_1, \ldots, e_n \). For example, \( [X > Y] \cdot 2 < X, 0, f(Y) \) . If \( X > Y \), variable \( X \) and variable \( Y \) of transitions \( t \) generate two items of elements, its type is \( < X, 0, f(Y) \) , otherwise generate empty multiset.

**Definition 3.4:** (Colour mapping) Assume \( C, C' \in \varepsilon \), colour mapping set from \( C \) to \( C' \) is defined as \( Map_{c, c'} \). \( Map_{c, c'} = \bigcup_{i=1}^{k} tup_i \) .
Colour mapping can be used as tuple joint, for example $tup \in map \Rightarrow map = tup + map'$. Colours mapping stands for arc labels on Coloured Petri Nets, which is the sum of the elements expression tuples.

For example, the arc expression $E(a) = \text{if } OK \text{ then } \text{business} \text{ else } \text{business} \text{ uses colour mapping and is expressed as } Map = \bigcup_{i=1}^{1.2} tup_i$, in which $tup_1 = [ok]; 1 < \text{business} >, tup_2 = [-ok]; 2 < \text{business} >$.

We introduce symbol $\ast$ to extend basic types. The same extensions can be found in colour domains. This definition assumes that this symbol does not belong to any basic types.

**Definition 3.5:** (Expanded colour domain) Assume $\delta \in \Delta$, the extended type $\delta'$ is $\delta \cup \{\ast\}$. Extended type set $\Delta'$ is $\{\delta' | \delta \in \Delta\}$. Suppose $C \in \zeta$, extended colour domain $C'$ is a Cartesian product $C \times \cdots \times C_{S(C)}$. Expanded colour domain set is expressed as $\zeta'$.

Considering extension of colour domain, this paper must modify the semantic of element expressions. If the sub-expression of function expression $e$ is evaluated as $\ast$, then $e$ is also $\ast$. Otherwise, its value does not change.

**Definition 3.6:** (Extended elements expression) Assume $C \in \zeta$, $\delta' \in \Delta$ and $e \in e_{C, \delta'}$, $e'$ is the extended element expression sets from $C'$ to $\delta'$. It can be defined as:

$$e'(c) = \begin{cases} if \ e = V, \ then \ c_i, \\ if \ e = (f, < e_{1} \ldots e_{n} >), \ then \\ \quad if \ \forall i \in [1 \ldots m], e_{i}'(c) \neq \ast \ then \ e(c) \\ \quad \text{else } \ast \end{cases}$$

This paper uses an extending class to enumerate all the elements of a class, such as $Unf_{Bool\cdot Bool}(< \ast, \text{true} >) = \{< \text{false}, \text{true} >, < \text{true}, \text{true} >\}$.

**Definition 3.7:** (Extending class) Assume $C \in \zeta$, $\delta \in \Delta$, the mapping from $\delta'$ to $P(\delta)$ is defined as $Unf_{\delta'}$, the mapping from $C'$ to $P(C)$ is defined as $Unf_{C'}$.

$$Unf_{\delta'}(e) = if \ e = \ast \ then \ \delta' \ else \ e$$

$$Unf_{C'}(< c_{1}, \ldots, c_{n} >) = Unf_{C'}(c_{1}) \times \cdots \times Unf_{C'}(c_{n})$$

We use $\succ_{C}$ to define each basic type or inclusion relation in colour domain $C$. For example, $< \text{true}, \ast > \succ_{\text{Bool}\cdot\text{Bool}} < \text{true}, \text{false} >$ holds true, but $< \text{true}, \ast > \succ_{\text{Bool}\cdot\text{Bool}} < \text{false}, \ast >$ does not hold. If $C \succ_{C} c'$, then $Unf_{C'}(c') \subseteq Unf_{C'}(c)$, that is $c'$ is the subclass of $c$.

**Definition 3.8:** (Inclusion relation) Assume $C \in \zeta$, the relationship $\succ_{C}$ between $C' \times C'$ is defined as $< c_{1}, \ldots, c_{n} > \succ_{C} < c'_{1}, \ldots, c'_{n} > \Rightarrow \forall i \in [1 \ldots n], c_{i} = \ast \lor c_{i} \succ_{C} c'_{i}$.

**Definition 3.9:** (Transition binding class) Assume $t \in T$, binding sets of transition are expressed as $C_{p}$, if $c_{i} \in C_{p}$, then $c_{i} \in C'$. Binding class of transition $t$ is represented as two-tuples $(t, C_{p})$.

For example, the binding situations that transition $t$ may occurs in include $< x = 1, y = 3 >$ and $< x = \ast, y = 2, z = 4 >$ then $C_{p} = \{< x = 1, y = 3, x = \ast, y = 2, z = 4 >\}$. Therefore, transition binding class is a set of transition binding elements.

**Definition 3.10:** (Place binding class) Assume $p \in P$, binding sets of place $p$ is expressed as $C_{p}$, if $c_{p} \in C_{p}$, then $c_{p} \in C'$. Binding classes of place $p$ represent a two-tuples $(p, C_{p})$.

For example, $(p, < x = 2, y = \ast >)$ refers to the token colour $< x = 2, y = \ast >$ stored in place. The token can be consumed by after transitions of places. Therefore, $c_{p}$ stands for the tokens that stored in $p$, which can also be represented as the binding that needed to consume.

3.2 Stubborn set state space construction method

3.2.1 Transition dependency graph for integrity verification

In this section, we begin to construct Transition Dependency Graph (TDG) from exception handling process in CPN model. On one hand, we propose the definition of TDG, using direction graph to express dependencies relation between transitions and sets of binding elements. The transitions support boundness analysis of particular place sets. On the other hand, by analysing model structure of exception handling process described by CPNs, we give a detailed description of TDG construction algorithms. Regarding its two functions $\text{reverseMap}_{(p, \ast)}(C_{p}, map)$ and $\text{capegoat}(t, C_{p}, m)$, we have proposed corresponding methods to solve the problem.
boundness of the specific place sets is only related to the binding elements that generate token or consume token in the places. We record the structure of transition sets and transition execution dependences and we call this structure TDG. TDG is a superset of stubborn sets which guides the occurrence of binding elements about exception generating, exception handling and exception throwing. TDG reserves the relevant state spaces related to integrity verification of exception handling.

Therefore, TDG is a directed graph. Node in the graph is corresponding to transition that is not substitute transition in hierarchical Coloured Petri Nets module, which means \( T_i = \{(t,s) \mid s \in S \land t \in T - T_{\text{sub}}\} \). There are directed arcs between two nodes and the occurrence of the source node \( T_i \) associated binding elements may cause the target node \( T_i \) enabling.

**Definition 3.11** (Transition dependency graph): TDG of hierarchical Coloured Petri Nets is a directed graph 

\[
TDG = (N_{TDG}, A_{TDG}), N_{TDG} = \bigcup_{i \in T} \{(t,s) \mid s \in S \land t \in T - T_{\text{sub}}\}
\]

is a set of nodes, \( A_{TDG} = \{(t_i, c_i, T_i) \in N_{TDG} \times C \times N_{TDG} \mid T_i \rightarrow t_i\} \) is a set of arcs.

In place–transition net, we can obtain dependence between transitions by analysing net structure. But in Coloured Petri Nets, to obtain dependence between transitions, we should not only analyse net structure, but also consider the colour mapping which is expressed as arc labels between places and transitions. Valmari (1991) proposed to extend Coloured Petri Nets model by enumerating all the transitions binding before state space detecting, remove the impact of transition dependence by colors in net. But the extension of Coloured Petri Nets costs too much, and may lead to an infinite scale Petri net.

The construction method of TDG is shown in Figure 1. The figure consists of three parts: Coloured Petri Nets, construction procedures, and TDG, showing Coloured Petri Nets model obtains the corresponding TDG after a series of construction procedures. TDG records the transition sets and transition dependences produced by specific sets of places to support integrity analysis of exception handling. Based on the thinking of solving define domain of a function according to range of the function values, the construction of TDG starts with specific place sets, reversely obtains transitions and transitions binding class which generate tokens in this place. Based on the above transitions and transitions binding classes, we can reverse to resolve place sets and place binding class sets that the transition firing needs. The above sets of places are specific sets of places, and we repeat such process until the calculating places belong to the normal business process.

TDG construction algorithm is a modified version of graph breadth-first search algorithm. First, use \( Nodes \) and \( Arcs \) respectively to represent the nodes and directed arcs in TDG (lines 1–2). \( PreSet \) stands for untreated sets of transition binding class nodes (line 3). \( PostSet \) stands for post-transitions binding class nodes \( (t_{post}, C_{t_{post}}) \) resulting in transitions binding class nodes enabling (Line 4). Through \( produceClasses \) calculation, we get the final calculated binding class sets of pre-transitions of exception thrown places, and inset the sets into \( PreSet \) (line 5). The algorithm loops until untreated transitions binding class doesn't exist. In each iteration of the loop (lines 6–24), the algorithm selects an unhandled transition binding class, and removes it from \( PostSet \) (lines 7–8). If transition \( t_{pre} \) in transitions binding class does not include in \( Nodes \), then \( t_{pre} \) is inserted into \( Nodes \) (lines 9–11). The algorithm inserts all post-transitions that make \( t_{pre} \) enabled and the corresponding binding classes into \( PostSet \) (line 12). \( PostSet \) records all the transitions and the corresponding binding classes that have dependency relationship with \( t_{pre} \), repeat recording \( PostSet \) elements of \( t_{pre} \) until \( PostSet \) is empty (lines 13–23). Takes any \( (t_{post}, C_{t_{post}}) \) from \( PostSet \) (lines 14–15), if \( t_{post} \) does not include in \( Nodes \), then inserts \( t_{post} \) in \( Nodes \) (lines 16–18).

**Figure 1** Construction method of TDG

- \( C_{t_{post}} \) represents the dependency relationship between \( t_{post} \) and \( t_{pre} \). Then the algorithm saves \( C_{t_{post}} \) into \( Arcs \) as the arcs from \( t_{post} \) to \( t_{pre} \) (line 19). If \( (t_{post}, C_{t_{post}}) \) does not include in \( PreSet \), insert \( (t_{post}, C_{t_{post}}) \) into \( PreSet \) (lines 20–22).

1: \( Nodes := \emptyset \)
2: \( Arcs := \emptyset \)
3: \( PreSet := \emptyset \)
4: \( PostSet := \emptyset \)
5: \( PreSet \leftarrow produceClasses((\text{throwPoint}, PM), C_{\text{(throwPoint,PM)}}) \)
6: while \( PreSet \neq \emptyset \) do
7: Select \( (t_{pre}, C_{t_{pre}}) \in PreSet \)
8: \( PreSet := PreSet - \{(t_{pre}, C_{t_{pre}})\} \)
9: if \( t_{pre} \notin Nodes \) then
10: \( Nodes := Nodes \cup \{t_{pre}\} \)
The idea is similar to giving a range of function values and the function body and solving definition domain of the function. Binding class is token types stored in the places. According to the arc expressions, we can find the sets of transition binding elements that can generate the types of tokens. The set describes dependency relation between Coloured Petri Nets transitions. This method also has certain limitations, such as the need of enumerating colour domain of place binding class. The element expression tuples used to describe the arc expressions need to convert to the formal of variables, and expressions with more complexity will not conduct reverse colour mapping.

Each pair $(p, t)$ of $P \times T$ defines a reverse colour mapping $\text{reverseMap}_{p,t}$. Assume the colour mapping of transit $t$ from $C$ to $C'$ is $map$, reverse colour mapping $\text{reverseMap}_{p,t}^{-1}(C_p, map)$ of $map$ calculates binding class of transition $t$, i.e. transition binding sets that produce the tokens of class $c_p$. Because colour mapping is the sum of the element expression tuples, the problem can be reduced to $\text{reverseMap}_{p,t}^{-1}(C_p, map) = \bigcup_{tup, map} \text{reverseTup}_{tup}(C_p, tup)$.

Solving steps of $\text{reverseTup}_{tup}(C_p, tup)$ are as follows:

The first step is to solve the transition binding $r$. We take one of the elements $c_p$ from $C_p$ (line 1-2), using the expanded colour domain $<*, \ldots, *>$ to initialise respective weights of unsolved binding $r$ (line 3). By searching for variables in $tup$, we modify the variables in $r$.

If variable $V_j$ occurs in the tuple position $i$, then replace $*$ of $r$ in $j$ position with the items of $c_p$, then replace $*\rightarrow$ of $i$ in $r$ with the items in position $i$ of $c_p$. For example:

\[
\text{reverseTup}_{tup}(<0,1,2>,<X,Z,f(Y)>) = <X = 0, Y = *, Z = 1 >
\]

The second step is to check the legality of $r$. When $e_i(r)$ of $tup$ in $i$ position is not $*$, and does not equal to $c_i$, for example $\text{reverseTup}_{tup}(<2,2>,<X,X+1>)$ appears. Transitions in this case do not generate $c_p$ by $tup$, thus it will be out of this iteration (lines 10-12). When the guard function of $tup$ is false, like $\text{reverseTup}_{tup}(<0,*,>, [X > 0]<X,Y>)$, it will be out of this iteration (lines 13-15).

Solving process of $\text{reverseTup}_{tup}(C_p, tup)$ is as follows:

1: while $C_p \neq \emptyset$ do
2: \hspace{1cm} $c_p \in C_p$, $C_p = C_p - \{c_p\}$
3: \hspace{1cm} Let $c_p = \langle c_{i_1}, \ldots, c_{i_n} >$
4: \hspace{1cm} Let $tup = [y] \cdot \alpha < e_{i_1}, \ldots, e_{i_n} >$
5: \hspace{1cm} $r \leftarrow <*, \ldots, *>$
6: \hspace{1cm} For $i \in [1, \ldots, n]$ do
7: \hspace{2cm} if $e_i = V_j$ and $r_j = *$ then $r_j = c_i$
8: \hspace{2cm} if $e_i = V_j$ and $r_j = *$ then $r_j = *$
9: \hspace{1cm} end for
10: if $\exists i \in [1, \ldots, n] e_i(r) \neq * \land c_i \neq * \land e_i(r) \neq c_i$ then
11: \hspace{1cm} Exit
i) Tokens stored in the place $p$, its colour coefficient is less than $\alpha$ coefficient in the tuple.

ii) The values produced by the expression $e_i$ of tuples in position $i$, are $\ast$, and are not equivalent to the values of $c_p$ in position $i$, such as the above $(p_2, \ast, false)$.

Case 2: If there is no tuple of the above types, we choose all the tokens that are consumed by binding class $c_i$. The token sets must be scapegoats of $c_i$. For example, the above $(p_1, <2, \ast>, (p_2, \ast, false))$.

3.2.2 Stubborn set state space construction based on TDG

The input of stubborn set construction algorithm includes current id of model $m$, TDG, output includes stubborn set $\text{Stub}$ of id $m$.

We first create a set of untreated binding elements named $\text{Calcu}$, and use all the enabled binding element sets to initialise $\text{Calcu}$ under identity $m$ (line 1). If TDG is traversed completely and there is no TDG containing a binding element, then the state space construction completes, and then returns directly (lines 3-5). Each binding element in untreated binding elements sets is iterative processed (line 6-15). If the transitions of binding elements are the part of nodes sets in TDG, and there is transition binding class $c_i$ including binding $b_i$ in TDG, then this binding element will be removed from untreated binding element sets and inserted into stubborn set $\text{Stub}$ (lines 7-9). In TDG, the traversed nodes and the $c_i$ are are marked. After the above setups, inserts this element into stubborn set $\text{Stub}$ (lines 19-21). If it does not exist, then inserts any element in $\text{Calcu}$ into stubborn set $\text{Stub}$.

$\text{STUBBORN} (m, \text{TDG}, \text{ProcessID})$

1: $\text{Calcu} := \{\text{all enabled binding elements in } m\}$

2: $\text{Stub} := \emptyset$

3: if $\text{TDG} = \emptyset$ then

4: return $\emptyset$

5: end if

6: For $(t_i, b_i) \in \text{Calcu}$ $i \in [1, \ldots, n]$ do

7: if $t_i \in N_{\text{TDG}}$ then

8: if $\exists c_i \in C_i, b_i \prec c_i$ then

9: $\text{Calcu} := \text{Calcu} - \{(t_i, b_i)\}$ and

10: $\text{Stub} := \text{Stub} \cup (t_i, b_i)$

11: $A_{\text{TDG}} = A_{\text{TDG}} - \{(t_i, c_i, \ast)\}$

12: end if

13: end if

14: end if

15: end for

Figure 2 Computing transition scapegoat of $t_1$

Suppose binding class $c_i =< X = 2, Y = \ast >$ of transition $t$:

i) If $m(p_1) =< 2, 3 >, m(p_2) =< 4, true >$, due to all binding of $c_i$ in the place $p_2$, there is no token using false as the second part, the scapegoat of $c_i$ in current state is $(p_2, \ast, false)$. 

ii) If $m'(p_1) =< 2, 3 >, m'(p_2) =< 4, false >$, $(q, <\ast, false>)$ is no longer the scapegoat of transition $t$ in the current state. The scapegoat of $c_i$ is $(p_1, <2, \ast>, (p_2, <\ast, false>))$.

Therefore, the purpose of the function $\text{scapegoat}(t, C_i, m)$ is to find relevant scapegoat sets in given id $m$ and transition binding class $(t, C_i)$. The function has two ways to solve:

Case 1: If all bounding $c_i$ of transition $t$ has unique scapegoats. Sufficient condition is when $\gamma(c_i) = true$, the tuples $[\gamma]x < e_1, \ldots, e_2 >$ in input arcs from $p$ to $t$ satisfy the following conditions:

i) Tokens stored in the place $p$, its colour coefficient is less than $\alpha$ coefficient in the tuple.
3.3 Exception handling integrity search based on stubborn set state space

We have converted the exception logic integrity verification problem into boundness analysis of a place set \( \text{ThrowPoint}_{\text{e}} \) of final exception thrown. In order to obtain the biggest integer bound of \( \text{ThrowPoint}_{\text{e}} \), we need to search all the nodes in state spaces, find the number of tokens \( |M(\text{ThrowPoint}_{\text{e}})| \) in place \( \text{ThrowPoint}_{\text{e}} \) in each node \( M \), and return the maximum value. The formal expression is as follows:

**Proposition 3.1:** Assume that \( SS = (N_s, A_{\text{e}}) \) is a finite state space of a service-oriented software exception handling model \( CPN_{\text{e}} \):

\[
\forall \text{ThrowPoint}_{\text{Set}}, \forall \text{ThrowPoint}_{\text{e}} \in \text{ThrowPoint}_{\text{Set}}, \text{meets} \text{ThrowPoint}_{\text{e}} \subseteq P \land (\text{ThrowPoint}_{\text{e}})^{'} = \{ \emptyset \}, \text{when and only when } \max\{ |M(\text{ThrowPoint}_{\text{e}})| \mid M \in N_s \} = 0, \text{ CPN}_{\text{e}} \text{ corresponding service-oriented software exception handling is integrated.}
\]

4 Related Work

4.1 Modelling and verification of exception handling

Petri net-based modelling and verification methods of exception handling include: Ana-Elena Rugina (Rugina et al., 2007) of University of Toulouse in France used AADL for system reliability modelling and proposed model transformation rules that convert the reliability of the model into GSPN. Dong et al. (2011) of Northwestern Polytechnical University in China described dynamic characteristics of interactions between components in AADL architecture system based on GSPN, analysis probable failure status. Literature (Zhu et al., 2011) designed an automatically generated runtime monitor from the BPEL description. A formal representation model based on Coloured Petri Nets was introduced to extract the service interaction behaviours from its description. The pattern mapping rules and related embedding, reduction and composition rules are also provided. In order to guarantee the reliability of service composition, East China University’s Fan Guisheng (Fan et al., 2013) proposed a reliable composition strategy and considered exception handling for services in the non-failure task. The related theories of Petri nets help to prove the effectiveness of the proposed method and analyse the state spaces size of reliable composition model. Richard Mrasek (Mrasek et al., 2015) from KIT Institute verified an industrial process efficiently by exploiting the structure of the high-level process schema. A new algorithm traversed the process structure tree and identifies the regions of the process that were relevant for verification of a given complex requirement. They also created a formal reduced representation of the process for each requirement based on Petri nets and used relevance function to feature a criterion for process-graph reduction.

Other modelling and verification methods include: Pereira (Pereira and de Melo, 2010) of Brazil São Paulo University used built-in Communicating Sequential Processes (CSPs) and predefined channels to coordinate exception handling in user-defined components. He verified concurrent exception handling related security attributes, asserted the security attributes, and used FDR to analyze validation. Wolf Zimmermann (Heike et al., 2013) proposed an approach for checking whether service protocols are obeyed in a service composition. Service protocols specified for a service legal sequences of operation calls. They abstracted exception handling based on directed graph, and verified whether the set of possible operation calls sequences to a stateful service is a subset of legal operation calls set specified by the protocol. Peter Csaba Olveczky (Bae et al., 2011) studied formal verification methods of AADL models based on behaviour annex. They proposed a formal semantics of AADL models in rewriting logic for a behavioural fragment, and specified a formal synchronous semantics for the fragment. They also embodied this semantics in a tool to simulate and verified models by LTL model checking. Marco Roveri (Bozzano et al., 2011) modelled features for random error behaviour based on a subset of AADL (SLIM Language), and provided a mapping onto networks of event-data automata to check correctness properties of dependability. Prakash Prabhu (Prabhu et al., 2011) presented a modular abstraction for capturing the interprocedural control flow induced by exceptions in C++, called the interprocedural exception control flow graph (IECFG), and transformed it into an exception-free program that was amenable for precise static analysis.

The above formal description and verification methods aim at different exception handling mechanisms. However, exception handling lacks integrity verification methods, which brings difficulty in guaranteeing that all exceptions have been handled for service-oriented software.

4.2 State space construction and reduction method base on Petri nets

Researchers have proposed various state space methods. They are mainly divided into: analysis of state space subsets and analysis of state space compression.

State space subset method generates partial state spaces of full state spaces. By reducing the number of independent crosses to reduce state spaces, so it can not cover the whole state spaces, but it is necessary to ensure that reduction does not affect requirements.
Verifying integrity of exception handling in service-oriented software

Stubborn set analysis method (Valmari and Hansen, 2010) refers to build stubborn set state spaces based on a specific stubborn set and analyse state attributes. Building stubborn set is crucial for stubborn set analysis method. The decisive factors of stubborn set construction includes: dependencies between transitions and desired verification properties. Hanifa Boucheneb (Boucheneb and Barkaoui, 2015) revisited stubborn set method in the context of POSETs by ignoring some firing order constraints of transitions. They established some practically sufficient conditions for stubborn sets to compute reduced state class graphs. They showed that the resulting reduced graph preserves deadlocks of the TPN the K-boundedness of places. Sweep line analysis method (Jensen et al., 2012) uses the system characteristics of a particular type state, by detecting state space fragments in specific system state, reduces the utilisation rate of memory during state space analysing.

State space compression construction method is about compressing whole state spaces. It declares equivalent states and only stores representative node of each collection. Therefore, it ignores a lot of state spaces, but increases the calculation of the decision of state compression. Its advantage is that it can cover the whole state spaces. Its representative methods include symmetrical analysis methods and equivalent analysis methods.

Symmetrical analysis method (Lorentsen and Kristensen, 2001) used the feature that a state space of symmetric system also has symmetrical characteristics, used Equivalence class (Christensen et al., 2001) to identify the symmetrical identification and symmetrical binding elements, in order to achieve symmetrical compression of state spaces. Equivalent analysis method is a kind of generalised symmetry analysis method, allowing more dynamic / generalised equivalent concept, so it can be used when system lacks symmetry in the system. For High-Level SPN state reduction, Marco Beccuti (Beccuti et al., 2015) automatically derived a reduced set of ordinary differential equations (ODEs) which mimic the system behaviour. Exploiting symmetrical properties of SSN representations is not only at the construction level, but also at that of the solution. For function-level reuse of legacy parallel systems, Jing Liu (Sun et al., 2012) proposed a system behaviour preduction approach based on trace-equivalent to generate an external behaviour equivalent model with smaller scale, and confirmed effectively the reuse contents in original parallel software.

5 Cases and experiment results

In this section, we take retry mode in exception handling for example, give integrity verification steps by TDG construction, and design experiments to verify the proposed method.

5.1 TDG construction of retry model

Assume place binding class \( \text{throwPoint}^{\text{invoke}}_c \), \( \text{throwPoint}^{\text{invoke}}_c \) in \( \text{throwPoint}^{\text{invoke}}_c \) for any \( c \in \mathbb{C}_{\text{throwPoint}^{\text{invoke}}} \) \( c \) is union type. Extend all types in \( c \), when \( c \) is union type, \( Unf(c) \) = \( \langle \text{CORR} = *, \text{uni} = \text{uninitializedVariable}, * \rangle \). \( \mathbb{C}_{\text{throwPoint}^{\text{invoke}}} \) = \( \{ \text{CORR} = *, \text{bind} = \text{bindingFault}, * \} \), \( < \text{CORR} = *, * \} \), \( < \text{CORR} = *, \text{notIntegrity} = \text{NOTIN}, * \} \), \( < \text{CORR} = *, \text{invokInteFault} = \text{notConnection}, * \} \), \( < \text{CORR} = *, \text{invokAlarmFault} = \text{cannotAlarm}, * \} \) through extending all the elements.

Bring \( \langle \text{throwPoint}^{\text{invoke}}_c \rangle \) into the formula as follow: \( \text{produceClasses}(p, \mathbb{C}_p) = \bigcup_{c \in \mathbb{C}_p} \langle \text{produceClasses}(p, c) \rangle \). \( (\text{throwPoint}^{\text{invoke}}_c) \) = \{ \text{rethrow} \} \text{EHPINVOKEThrow} \}, \text{rethrow} \} \text{Elhate Return} \} .

When \( t = \text{rethrow} \} \text{EHPINVOKEThrow} \):

Create a node in TDG (reThrow, EHPINVOKEThrow), as shown in Figure 3.

Figure 3 Retry exception handling model (see online version for colours)
reverseMap\left(C_{\text{throwPoint}}\cup Map\right) = \bigcup_{\text{tmap}} \text{reverseTup}_{\text{throwPoint}}\left(C_{\text{throwPoint}}\cup Map\right)

\left(C_{\text{throwPoint}}, \text{tmap} \right) = \text{reverseTup}_{\text{throwPoint}}\left(C_{\text{throwPoint}}, \text{tmap} \right), < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \left\{ < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\}.

Therefore, \(\left(t, C_{\text{throwPoint}}\cup Map\right)\) into Figure 3.

\(\text{reverseTup}_{\text{throwPoint}}\left(C_{\text{throwPoint}}, \text{tmap} \right) = \text{reverseTup}_{\text{throwPoint}}\left(C_{\text{throwPoint}}, \text{tmap} \right), < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \left\{ < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\}.

\text{solve} \left(C_{\text{throwPoint}}, \text{tmap} \right) = \left\{ < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\}.

\text{Substitute} \left(t, C_{\text{throwPoint}}\cup Map\right) \text{into Figure 3.}

\text{Create a node} \left(t, \text{input, Invoke}\right) \text{in TDG, as shown in Figure 3.}

\text{Put} \left(t, \text{input, Invoke}\right) \text{and} \left(t, \text{throwPoint}\right) \text{into the reverse colour mapping function:}

\text{reverseMap}\left(C_{\text{throwPoint}}\cup Map\right) = \bigcup_{\text{tmap}} \text{reverseTup}_{\text{throwPoint}}\left(C_{\text{throwPoint}}\cup Map\right)

\left(C_{\text{throwPoint}}, \text{tmap} \right) = \text{reverseTup}_{\text{throwPoint}}\left(C_{\text{throwPoint}}, \text{tmap} \right) = \left\{ < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\}.

\text{Nodes} \left(t, \text{input, Invoke}\right) \text{in TDG to the nodes} \left(t, \text{throwPoint}\right) \text{and add tags} \left< \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\> \text{to directed arcs.}

\text{When} \ t = \text{reThrow} \text{in TDG, add the node} \left(t, \text{throwPoint}\right) \text{to the nodes} \left(t, \text{throwPoint}\right) \text{and} \left(t, \text{Invoke}\right) \text{in TDG.}

\text{reverseMap}\left(C_{\text{throwPoint}}\cup Map\right) = \bigcup_{\text{tmap}} \text{reverseTup}_{\text{throwPoint}}\left(C_{\text{throwPoint}}, \text{tmap} \right)

\left(C_{\text{throwPoint}}, \text{tmap} \right) = \text{reverseTup}_{\text{throwPoint}}\left(C_{\text{throwPoint}}, \text{tmap} \right) = \left\{ < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\}.

\text{Thus, set} \left(t, C_{\text{throwPoint}}\cup Map\right) = \left\{ < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\} \text{and} \left(t, C_{\text{throwPoint}}\cup Map\right) = \left\{ < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\}.

\text{Enable} \left(C_{\text{throwPoint}}\cup Map\right) = \bigcup_{\text{tmap}} \text{enableClasses}\left(t, C_{\text{throwPoint}}\cup Map\right) = \left\{ < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\}.

\text{Solve} \left(t, C_{\text{throwPoint}}\cup Map\right) = \left\{ < \text{CORR} = \#1 \text{uniFault}, \text{uninitializedVariable f int egrityWeb} > = \right\} \text{and add directed arcs tagged} \left< e > = \right\> \text{to nodes} \left(t, \text{input, Invoke}\right) \text{in TDG.}

\text{When} \ p = \text{output} \text{in TDG, add a directed} \left< e > = \right\> \text{to the label} \left< e > = \right\> \text{and} \left< e > = \right\> \text{in TDG.}

\text{Add nodes} \left(t, \text{output, Invoke}\right) \text{to the nodes} \left(t, \text{throwPoint}\right) \text{and} \left(t, \text{Invoke}\right) \text{in TDG.}

\text{We constructed an experimental environment to confirm the effectiveness of our method. We extracted three exception handling scenes from a real-world service-oriented system to model exception handling. We extended automatical functions of analysis and verification provided by CPN tools 4.0, and implemented exception handling integrity verification.}

\text{The scenes come from part of Comprehensive Disaster Reduction Information Service Application System (CDRISAS) in our previous project. The normal processes of the system include six services: when the disaster occurred, operators firstly create an emergency task (service 0), then they choose to use the workgroup templates (service 2) or re-create a working group (service 1) for the current emergency tasks, and then create a task for the working group (service 3), and finally system starts the current processes (services 4) and generates events (service 5).}
Verifying integrity of exception handling in service-oriented software

Various service nodes of the system are likely to fail and generate exceptions. New exceptions may be added because of exception propagating and transforming in service-oriented software. We set up 10, 20, 30 exceptions for every scene respectively. We also assume five kinds of exception handling behaviours: retry, alternate, wait, skip and notice. For example, one possible exception handling mode is: if service 0 fails, the system selects retry, and the retry will attempt three times at most; if service 2 fails, the system selects service 1 as an alternative choice, and wait after services 1 and 2 both fail; the system selects to skip over after service 3 fails; if skip also fails, then selects notification; the system selects a notification either service 4 or 5 fails. The results of exception handling and integrity verification are as follows in Table 1.

Figures 5–7 demonstrate the results of state spaces reductions.

Table 1 Results of exception handling integrity verification

<table>
<thead>
<tr>
<th>Scene</th>
<th>Exceptions sets</th>
<th>Triggered exceptions</th>
<th>Unhandled exceptions verified</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>retry 2</td>
<td>retry exceptions 1</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>Alternate 0</td>
<td>Alternate exceptions 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait 17</td>
<td>wait exceptions 11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>skip 7</td>
<td>skip exceptions 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>notice 39</td>
<td>notice exceptions 13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total 65</td>
<td>total exceptions 28</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>retry 9</td>
<td>retry exceptions 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alternate 25</td>
<td>Alternate exceptions 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait 21</td>
<td>wait exceptions 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>skip 18</td>
<td>skip exceptions 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>notice 36</td>
<td>notice exceptions 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total 46</td>
<td>total exceptions 24</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>retry 78</td>
<td>retry exceptions 36</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Alternate 199</td>
<td>Alternate exceptions 101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait 278</td>
<td>wait exceptions 34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>skip 345</td>
<td>skip exceptions 57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>notice 714</td>
<td>notice exceptions 268</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total 1614</td>
<td>total exceptions 496</td>
</tr>
</tbody>
</table>

Figure 4 TDG of boundedness analysis
6 Conclusion

In this paper, we have verified integrity of exception handling for service-oriented software based on hierarchical CPN. First, we have introduced and extended a service-oriented software exception handling model and its integrity. We have added formal description of exception handling actions and converted the issue of integrity verification into boundedness analysis of specific places. We use Petri net state space methods to implement integrity verification of service-oriented software exception handling. To avoid the problem of state space explosion, we have highlighted a construction method of state space stubborn set based on TDG. By searching all nodes of state spaces and returning the number of tokens in nodes, we have achieved the integrity verification of service-oriented software exception handling. According to experimental result, the size of state spaces is reduced and integrity verification for service-oriented software is accomplished. Although the method in this paper is for integrity verification of exception handling model, its design thinking is universal. It provides some experiences for verifying other actions or attributes. In the next stage, we will continue to deepen formal description of exception handling to support more accurate formal semantics in service-oriented software, and try to achieve integrity verification of exception handling automatically.

Acknowledgements

This work was supported by the National Natural Science Foundation of China under Grant No.61373038, No.61272108, and the Foundation of Guangxi Key Laboratory of Trusted Software No.kx201534.

References


Verifying integrity of exception handling in service-oriented software


Websites


