FROM ENABLING TO ENSURING GRID WORKFLOWS

Junwei Cao[1,3,*], Fan Zhang[2], Ke Xu[2], Lianchen Liu[2,3], and Cheng Wu[2,3]
[1]Research Institute of Information Technology, Tsinghua University
[2]National CIMS Engineering and Research Center, Tsinghua University
[3]Tsinghua National Laboratory for Information Science and Technology
[*]Corresponding email: jcao@tsinghua.edu.cn

Abstract

Grid workflows are becoming a mainstream paradigm for implementing complex grid applications. In addition to existing grid enabling techniques, various grid ensuring techniques are emerging, e.g. workflow analysis and temporal reasoning, to probe potential pitfalls and errors and guarantee quality of services (QoS) at a design phase. A new state π calculus is proposed in this work, which not only enables flexible abstraction and management of historical grid system events, but also facilitates modeling and verification of grid workflows. Some typical patterns in grid workflows are captured and both static and dynamic formal verification issues are investigated, including structural correctness, specification satisfiability, logic satisfiability and consistency. A grid workflow modeling and verification environment, GridPiAnalyzer, is implemented using formal modeling and verification methods proposed in this work. Performance evaluation results are included using a grid workflow for gravitational wave data analysis.

1 Introduction

1.1 Grid workflow QoS

Advance in technology has made collections of internet-connected computers a viable computational platform. Grids connecting geographically distributed resources have become a promising infrastructure for solving large problems. The definition of Grids has been redefined over time. Initially Grids were defined as an infrastructure to provide easy and inexpensive access to high-end computing [1]. Then, it was refined in [2] as an infrastructure to share resources for collaborative problem solving. More recently, in [3] the Grid definition evolves into an infrastructure to virtualize resources and enable their use in a transparent fashion.

Grid workflows [4], a composition of various grid services according to prospective processes, have become a typical paradigm for problem solving in various e-Science domains [5], e.g. gravitational wave data analysis [6]. With increasing complexity of e-Science applications, how to implement reliable and trustworthy grid workflows according to specific scientific criteria is becoming a critical research issue. In addition to existing grid *enabling* techniques, e.g. job scheduling, workflow enactment and resource locating, various grid *ensuring* techniques are developed [7], e.g. data flow analysis and temporal reasoning.

Issues of quality of service (QoS) are of increasing importance to the success of those Grid-based applications. As defined by I. Foster in the three point checklist of the Grid [8], the Grid has to deliver to nontrivial qualities of service, relating for example to response time, throughput, availability, and security, and/or co-allocation of multiple resource types to meet complex user demands. This requirement is especially pronounced in experimental science applications such as the National Fusion Collaboratory [9] and NEESgrid [10]. Enabling such interactions on the Grid requires two related efforts: (1) the development of sophisticated resource management strategies and algorithms and (2) the development of protocols enabling structural negotiation for the use of those resources.

Most of existing research on grid workflow QoS is related to task scheduling. In the work described in [11], application performance prediction is coupled with genetic algorithms for workflow management and scheduling with consideration of makespans and job deadlines. QoS guided min-min heuristic for grid task scheduling is also proposed in [12]. Similar work can also be found in [13] and [14] for QoS aware grid workflow scheduling using performance prediction and optimization. In the grid standard organization, Global Grid Forum, a WS-Agreement model is proposed and defined in [15]. This provides an infrastructure to agreement-based application like [16] and [17], within which QoS can be negotiated and obtained.

While all of above in common is that they show how task can be scheduled to improve efficiency of grid workflows, this work is dedicated to ensuring mechanisms on workflows as a whole. All of services in a workflow are guaranteed without redundancy and collision. Also how to make sure all services in a workflow is reachable and terminatable is another concern in this work. All these issues are modeled, verified and finally implemented using our environment, *GridPiAnalyzer*.

1.2 Grid workflow verification

As mentioned above, it is significant for grids to implement large scale heterogeneous resource sharing and accessing. How to ensure the correctness of design and implementation of grid workflows is a critical task. Though it is widely recognized that corporation of grid workflows are important, most of those research work are focused on grid enabling techniques, e.g. automatic execution, service binding and transaction processing. In the field of grid workflows, formal semantics, business logic verification and improving of verification performance needs to be solved. Obviously, these formal verification techniques can ensure the correctness of workflows as a whole and guarantee the fulfilling of users' demands.

These intrinsic characteristics provide several challenges to formal verification:

- Difference in professional domains
- Complexity of applications
- Non-formalism semantics of grid workflows
- Diversity in grid workflows models
- Uniqueness of grid workflow criteria
- Dynamicity of grid environments

IEEE defines correctness as: "……free from faults, meeting of specified requirements, and meeting of user needs and expectations" [18], and formal verification as: "it is mathematical verification methods to test whether those system model can meet requirements" [19]. Requirements here can be interpreted from two aspects. It can be restraints from the system model or business logics of users' expectations. According to definitions mentioned above, the article includes four aspects:

- Structure verification
- Verification of semantic restraints in grid workflows
- Verification of users' demands
- Consistent verification of business logics

The following problems have to be solved to verify above issues:

- Formal theory and methods for grid workflow criteria
- Formal semantics for existing grid workflow criteria
- Dynamic/static verification methods
- Implementation of a grid workflow modeling and analysis environment

1.3 Grid workflow modeling

Many models are introduced as grid workflows become indispensable component of grid networks. Different models have different descriptions and semantics. From different application domains, grid workflows can be categorized as follows:

- XML-based tags, e.g. GridAnt [20], BPEL4WS [21] and Gridbus workflows [22].
- Visual languages, e.g. Triana [23], JOpera [24] and BPEL visual modeling.
- Customized script languages, e.g. Condor [25] DAGMan and Glue [26].

Different model specifications increase the complexity among various grid workflows. The integration of web and grid technologies is a clear trend since web service standards, e.g. Web Services Resource Framework (WSRF), are emerging. What's more, as BPEL4WS is gradually becoming the standard web execution language, more work is being related to the extension of BPEL4WS based on WSRF.

The motivation of our work is not to redefine a new model for grid workflows but rather try to find and propose a formal modeling and verification tool that works well with grid workflows. And hopefully the following can be achieved:

- Define critical characteristics and operations of grid workflows as well as bring out exact execution semantics of service interactions.
- Propose a uniform semantic basis as a bottom line for typical grid workflows.
- Verify grid workflows completely, automatically and effectively.

2 State π calculus

2.1 Introduction to π calculus

π calculus [27] was initially introduced by Milner's work for modeling state/action hybrid systems since it is intrinsically mobile and combinable. Nowadays, this tool is efficiently used in the description of open communication systems and web/grid workflows as described in [28] and [29]. The syntax of π calculus is as follows:

$$P ::= \sum_{i=1}^{n} \alpha_i.P_i \mid (new\, x)P \mid !P \mid P \mid Q \mid \phi P \mid A(y_1,...,y_n) \mid 0$$
$$\alpha_i ::= \bar{x} < \tilde{y} > \mid x(\tilde{y}) \mid \tau$$
$$\phi ::= [x = y] \mid \phi \wedge \phi$$

The fundamental concept of π calculus is the names, which are used to express atomic interactive actions in a system. A system in π calculus evolves through the operators including *composition* '|', *choice* '+', *guard* '.', *match* '[]', *restriction* 'new' and *replication* '!'.

- An output action ($\bar{x} < \tilde{y} > .P$): This means outputting $\tilde{y}$ through $\bar{x}$ with system behaviors evolved into *P*. For example, in a communication system $\bar{x}$ can be considered as an output port and $\tilde{y}$ the output data.
- An input action ($x(\tilde{y}).P$): Intuitively, it means inputting $\tilde{y}$ through *x* with system behaviors evolved into *P*.
- A silent action ($\tau.P$): The system behavior evolves into *P* with internal actions instead of interactions with the environment.
- A composition (*P*|*Q*): Processes *P* and *Q* are independent, or synchronize with each other via an identical port.
- Choice (*P*+*Q*): Unpredictable execution of *P* or *Q*.
- March ([*x*=*y*]*P*): If *x* matches *y*, the system behavior evolves into *P*. Otherwise no actions happen.
- Restriction ((*new x*) *P*): *x* is a new name within the process *P*.
- Replication (!*P*): An infinite composition of process *P*.

Process algebras like π calculus have an explicit description of system behaviors and interactions, but state models are implicitly. For life-cycle management of system states, it is needed to enable modeling of:

- creation and destruction of states;
- access and update of states;
- association of states with system behaviors.

2.2 State π calculus: models and operations

To address issues mentioned above, state π calculus is proposed in this work where a state *S* is defined as a finite set of system propositions *PROP*.

Definition 1 (system proposition). A system proposition *PROP*=(*ident*, *set*) ranged over a universe *D* is a pair, where *ident* is a unique identifier of *PROP* and *set* is the set of all valuations that make *PROP* to be true($set \in D$).

Consequently, a state is $S = \{p_1, ..., p_n\}$, where p_i (i=1,…, n) is the system proposition *PROP* ranged over *D*. To enhance the capability for the states to express their relations among different components in a system, in state π calculus the identifier *ident* is defined by the following hierarchical structure: *ident::=atom* | *atom.ident*. Here *atom* indicates a symbolic constant value and '.' indicates a separator for the *atoms*. Consequently, a prefix/suffix relation is used to define the hierarchical structure of *ident*:

$$prefix : ident \times ident' \to ident'' \qquad prefix(ident, ident') = ident'' \text{ IFF } ident = ident''.ident'$$
$$suffix : ident \times ident' \to ident'' \qquad suffix(ident, ident') = ident'' \text{ IFF } ident = ident'.ident''$$

For example, a state *S={(AvailableSrv, {Srv$_1$, Srv$_2$}), (Srv$_1$.Status, {Active}), (Srv$_2$.Status, {Input_Pending})}* can be used to indicate that in state *S*, there are two available services *Srv$_1$* and *Srv$_2$* in the system, where *Srv$_1$* is running and *Srv$_2$* is waiting for its input. The identifier *Status* is the suffix of *Srv$_1$* and *Srv$_2$*. To complete the static semantics of states and system propositions, three functions are further defined:

- *range*: *PROP→set* returns the set of all constants that make *PROP* true;
- *eval: PROP×valueset→{true, false}* determines whether *PROP* is true for a given set of values;
- *proposition: S×ident→{p$_1$, ..., p$_m$}* returns corresponding system propositions given an identifier. Since semantics of *range* and *eval* are quite straightforward, here we focus on the implementation of *proposition*. Note that due to the hierarchical structure of identifiers *ident*, the function of *proposition* should also be able to get all system propositions identified by the prefix of an *ident*.

$$\begin{array}{ll}
proposition(S, ident) = p & \text{if } \exists p = (ident, set) \in S \\
proposition(S, *) = \{p_1, ..., p_k\} & \text{if } \{p_1, ..., p_k\} = S \\
proposition(S, ident.*) = \{p_1, ..., p_m\} & \text{if } \{p_1, ..., p_m\} \subset S, \\
 & \forall p \in \{p_1, ..., p_m\}, \exists\ ident' \text{ s.t. } suffix(p, ident) = ident' \\
proposition(S, *.ident) = \{p_1, ..., p_n\} & \text{if } \{p_1, ..., p_n\} \subset S, \\
 & \forall p \in \{p_1, ..., p_n\}, \exists\ ident' \text{ s.t. } prefix(p, ident) = ident'
\end{array}$$

Besides static definition of system states and proposition, dynamic operational semantics of the creation, destruction and update of states based on system actions have also to be defined. Different from original π calculus, a group of state operators are introduced into the syntax of state π calculus. State π calculus aims to create dynamic association between states and actions in π calculus via state operators. Creation, destruction and update of states can thus be enabled by integrating operational semantics of original π calculus and semantics of state operators. Therefore, state π calculus reuse existing properties and analysis techniques in π calculus instead of revising the core of π calculus.

$$\begin{array}{rl}
P & ::= \sum_{i=1}^{n} \alpha_i \{StateExp\}.P_i \mid (new\ x)P \mid !P \mid P \mid Q \mid \phi P \mid A(y_1, ..., y_n) \mid 0 \\
\alpha_i \{StateExp\} & ::= \overline{x} < \widetilde{y} > \{StateExp\} \mid x(\widetilde{y})\{StateExp\} \mid \tau\{StateExp\} \\
\phi & ::= [x = y] \mid eval(Prop, valueset) \mid \phi \wedge \phi \\
StateExp & ::= (StateOp, S) \mid StateExp, StateExp \\
StateOp & ::= + \mid - \mid ++ \mid -- \\
S & ::= (iden, trueset) \\
iden & ::= x \mid iden.iden
\end{array}$$

As shown above, in state π calculus each input/output action can be associated with multiple state expressions *StateExpr*. *StateExpr* depicts possible state operations *StateOp* that a system action can do to a state. The choice operator '?' is also support in *StateExpr*. Consequently, the expression [*ConditionExpr*]*StateExpr?StateExpr* describes that when the condition *ConditionExpr* is (not) satisfied, a system action will be associated with the state expression of the former (latter) *StateExpr*. Four state operators are provided in state π calculus: state creation (+), state destruction (−), state association (++), and state removal (−−). An additional operation of state updates is also included in the state creation (+) operator. To be more intuitive, each state operator can be regarded as an association relation between states: $\mathfrak{R}$: $SysState \times StateOp \times S \rightarrow SysState$. That is, a new system state is determined by the current system state (*SysState*) and the target state operation ($StateOp \times S$). In state π calculus, a system state is the set of all existing states and their values in the system. Therefore, semantics of state operators are very essential to state π calculus, which define the association between system actions and states and life-cycles of system states.

Definition 2 (irrelevant states and conflict states). Two states S_1 and S_2 are *conflict* ($S_1 \Diamond S_2$) if $\exists p_1=(ident_1,set_1) \in S_1$ and $p_2=(ident_2,set_2) \in S_2$ s.t. $ident_1=ident_2$. Meanwhile system propositions p_1 and p_2 are *overlapped* ($p_1 \Diamond p_2$); S_1 and S_2 are *irrelevant* ($S_1 \nabla S_2$) if they are not conflict.

Definition 3 (state preordering). For any two nonconflict states S_1 and S_2, $S_1 \prec S_2$ if $\forall p=(ident,set) \in S_1$, $\exists p'=(ident',set') \in S_2$, s.t. $ident=ident'$ and $set \subset set'$; $S_1=S_2$ if $S_1 \prec S_2$ and $S_2 \prec S_1$.

Definition 4 (well-formed states). A state S is well-formed, iff the following two conditions are satisfied: (1) $\forall p_1, p_2 \in S$, there is no $p_1 \Diamond p_2$; (2) $\forall p=(ident,set) \in S$, *set* is not empty.

Two propositions in a well-formed state do not conflict and each proposition in the state is not always false. Given the current system state $SysState=\{p_{11},\ldots,p_{1n}\}$, and the state associated with the state operator *StateOp* $S=\{p_{21},\ldots,p_{2m}\}$, formal semantics of each state operator is provided as follows, defined over well-formed states.

1. State Creation (+): Define a new state and overwrite the existing one in the current system;

$$CREATE \qquad \frac{SysState \nabla S}{+S = \{p_{11},\ldots,p_{1n},p_{21},\ldots,p_{2m}\}}$$

$$UPDATE \qquad \frac{SysState \Diamond S \quad \exists p_{1i} \in SysState, p \in S \text{ s.t. } p_{1i} \Diamond p}{+S = \{p_{11},\ldots,p_{1i-1},p,p_{1i+1},\ldots,p_{1n},p_{21},\ldots,p_{2m}\}}$$

2. State Destruction (−): Remove a specific state from current system states;

$$DESTROY\_VOID \qquad \frac{SysState \nabla S}{-S = \{p_{11},\ldots,p_{1n}\}}$$

$$DESTROY \qquad \frac{SysState \Diamond S \quad \exists p_{1i} \in SysState, p \in S \text{ s.t. } p_{1i} \Diamond p}{-S = \{p_{11},\ldots,p_{1i-1},p_{1i+1},\ldots,p_{1n}\}}$$

3. State Association (++): Insert a state in current system states;

$$INSERT\_VOID \quad \frac{SysState \nabla S}{++S = \{p_{11}, \ldots, p_{1n}, p_{21}, \ldots, p_{2m}\}}$$

$$INSERT \quad \frac{SysState \Diamond S \qquad \exists p_{1i} = \in SysState, p \in S \text{ s.t. } p_{1i} \Diamond p}{\begin{array}{c} ++S = \{p_{11}, \ldots, p_{1i-1}, p_{1i}', p_{1i+1}, \ldots, p_{1n}, p_{21}, \ldots, p_{2m}\} \\ range(p_{1i}') = range(p_i) \cup range(p) \end{array}}$$

4. State Removal (−−): Remove the specific propositions from a state in the current system states;

$$REMOVE\_VOID \quad \frac{SysState \nabla S}{--S = \{p_{11}, \ldots, p_{1n}\}}$$

$$REMOVE\_SHALLOW \quad \frac{\begin{array}{c} SysState \Diamond S \qquad \exists p_{1i} = \in SysState, p \in S \text{ s.t. } p_{1i} \Diamond p \\ range(p_{1i}) / range(p) \neq \phi \end{array}}{\begin{array}{c} --S = \{p_{11}, \ldots, p_{1i-1}, p_{1i}', p_{1i+1}, \ldots, p_{1n}, p_{21}, \ldots, p_{2m}\} \\ range(p_{1i}') = range(p_i) / range(p) \end{array}}$$

$$REMOVE\_DEEP \quad \frac{\begin{array}{c} SysState \Diamond S \qquad \exists p_{1i} = \in SysState, p \in S \text{ s.t. } p_{1i} \Diamond p \\ range(p_{1i}) / range(p) = \phi \end{array}}{--S = \{p_{11}, \ldots, p_{1i-1}, p_{1i+1}, \ldots, p_{1n}, p_{21}, \ldots, p_{2m}\}}$$

As previously mentioned, above semantics form a basis for implementation of the state relation $\Re$, i.e.,

$$\Re : SysState \times StateOp \times S \rightarrow SysState'$$

$$\Re : (SysState, StateOp, S) = SysState'$$

, where *SysState*' is determined by the above 9 semantic rules.

2.3 State π calculus: extended operational semantics

In this section we integrate state operators with operational semantics of original π calculus, which leads to extended operational semantics for state π calculus. This interprets how system states and actions are mutually operated. Traditionally the behavior of π calculus is modeled using a standard Labeled Transition System (LTS). However, for modeling and reasoning of state/action hybrid systems, LTS should be extended to model both system actions (i.e. transition labels) and system states (i.e. state labels). Typical examples of these extensions can be found in the Labeled Kripke Structures [30] and the Doubly Labeled Transition Systems [31]. A *State Label Transition System* (SLTS) is proposed in this work for interpreting behaviors of state π calculus.

Definition 5 (state labeled transition system). An SLTS $(S, M, \{\xrightarrow{a\{StateExpr\}} \mid a \in M\})$ consists of a set *SP* of state/process pairs, a set M of transition labels, and a set $\{\xrightarrow{a\{StateExpr\}}\}$ of transitions $\xrightarrow{a\{StateExpr\}} \subseteq S \times S$ where $a \in M$.

In an SLTS, a transition is represented as $(P, SysState) \xrightarrow{a\{StateExpr\}} (P', SysState')$. This means the current process and state of the system is *P* and *SysState*, and by executing the action *a* associated with the state expression of *StateExpr*, the system process evolves to *P'* and the system state is updated to *SysState*'. According to SLTS, a static association *transState* can be defined between the system state *SysState* and its possible modification (*StateExpr*):

$$transS:\ SysState \times StateExpr \rightarrow SysState'$$

$$transS(SysState, StateExpr) = \begin{cases} \Re(SysState, Op, S) & StateExpr = (Op, S) \\ SysState & StateExpr \texttt{ is null} \end{cases}$$

Consequently, the extended operational semantics of state π calculus is defined below based on the early transitional semantics [32] of π calculus.

$$OUT \quad \frac{}{(\bar{x}<y>\{\varphi\}.P,\delta)\xrightarrow{\bar{x}<y>\{\varphi\}}(P,transS(\delta,\varphi))} \qquad INP \quad \frac{}{(x(z)\{\varphi\}.P,\delta)\xrightarrow{x(y)\{\varphi\}}\{y/z\}(P,transS(\delta,\varphi))}$$

$$TAU \quad \frac{}{(\tau\{\varphi\}.P,\delta)\xrightarrow{\tau\{\varphi\}}(P,transS(\delta,\varphi))} \qquad SUM-L \quad \frac{(P,\delta)\xrightarrow{\alpha}(P',\delta')}{(P+Q,\delta)\xrightarrow{\alpha}(P',\delta')}$$

$$MAT \quad \frac{(\alpha.P,\delta)\xrightarrow{\alpha}(P',\delta')}{([\phi]\alpha.P,\delta)\xrightarrow{\alpha}(P',\delta')} \qquad \phi \text{ means } x=x \text{ or } eval(Prop, valueset)=true$$

$$PAR-L \quad \frac{(P,\delta)\xrightarrow{\alpha}(P',\delta')}{(P\,|\,Q,\delta)\xrightarrow{\alpha}(P'\,|\,Q,\delta')} \qquad bn(\alpha)\cap fn(Q)=\varnothing$$

$$COMM-L \quad \frac{(P,\delta)\xrightarrow{\bar{x}<y>\{\varphi\}}(P',transS(\delta,\varphi)) \quad (Q,\delta')\xrightarrow{x(y)\{\varphi'\}}(Q',transS(\delta',\varphi'))}{(P\,|\,Q,\delta'')\xrightarrow{\tau\{\varphi,\varphi'\}}(P'\,|\,Q',transS(transS(\delta'',\varphi),\varphi'))}$$

$$CLOSE-L \quad \frac{(P,\delta)\xrightarrow{\bar{x}<z>\{\varphi\}}(P',transS(\delta,\varphi)) \quad (Q,\delta')\xrightarrow{x(z)\{\varphi'\}}(Q',transS(\delta',\varphi'))}{(P\,|\,Q,\delta'')\xrightarrow{\tau\{\varphi,\varphi'\}}(new\ z)(P'\,|\,Q',transS(transS(\delta'',\varphi),\varphi'))} \qquad z\notin fn(Q)$$

$$RES \quad \frac{(P,\delta)\xrightarrow{\alpha}(P',\delta')}{((new\ z)P,\delta)\xrightarrow{\alpha}((new\ z)P',\delta')} \qquad z\notin n(\alpha)$$

$$OPEN \quad \frac{(P,\delta)\xrightarrow{\bar{x}<z>\{\varphi\}}(P',transS(\delta,\varphi))}{((new\ z)P,\delta')\xrightarrow{\bar{x}<z>\{\varphi\}}(P',transS(\delta',\varphi))} \qquad z\neq x$$

$$REP-ACT \quad \frac{(P,\delta)\xrightarrow{\alpha}(P',\delta')}{(!P,\delta)\xrightarrow{\alpha}(P'\,|\,!P,\delta')}$$

$$REP-COMM \quad \frac{(P,\delta)\xrightarrow{\bar{x}<y>\{\varphi\}}(P',transS(\delta,\varphi)) \quad (P,\delta')\xrightarrow{x(y)\{\varphi'\}}(P'',transS(\delta',\varphi'))}{(!P,\delta'')\xrightarrow{\tau\{\varphi,\varphi'\}}((P'\,|\,P'')\,|\,!P,transS(transS(\delta'',\varphi),\varphi'))}$$

$$REP-CLOSE \quad \frac{(P,\delta)\xrightarrow{\bar{x}<z>\{\varphi\}}(P',transS(\delta,\varphi)) \quad (P,\delta')\xrightarrow{x(z)\{\varphi'\}}(P'',transS(\delta',\varphi'))}{(!P,\delta'')\xrightarrow{\tau\{\varphi,\varphi'\}}((new\ z)(P'\,|\,P'')\,|\,!P,transS(transS(\delta'',\varphi),\varphi'))} \qquad z\notin fn(P)$$

In above semantics, φ and δ are shortcut notations for *StateExpr* and *SysState* respectively; α denotes an arbitrary action in state π calculus; *fn* and *bn* are used to indicate the set of all free names and bounded names. Note that state π calculus does not tend to change the fundamental definition of Structural Congruence in π calculus, and reduction rules can also be extended similarly for state π calculus. Therefore, above operational semantics in state π calculus can be regarded as a further extension to the ones in π calculus for integrating system states with actions and management of these states.

2.4 State bi-simulation

Bi-simulation analysis is an important tool in process algebras to define process equivalence. In state π calculus, system states and their changes need to be further considered into the original strong (weak) bi-simulation relation in π calculus to define (observable) behavior equivalence between state/action hybrid systems. Denote $\Rightarrow^{\tau}$ to be a transition sequence triggered by invisible action τ. Denote $\Rightarrow^{a}$ and $\Rightarrow^{\tau}$ to be a transition sequence triggered by arbitrary action a where $a \neq \tau$ and $a = \tau$ respectively; Denote $\Rightarrow$ to be either $\Rightarrow^{a}$ or $\Rightarrow^{\tau}$ and $\Rightarrow^{a}$ to be the abbreviation for $\Rightarrow \xrightarrow{a} \Rightarrow$. A hybrid bi-simulation is defined below as a bi-simulation relation which considers both system states and actions.

Definition 6 (hybrid Bi-simulation). A symmetric binary relation R is a strong (weak) hybrid bi-simulation relation, iff for any $(P, SysState_P)R(Q, SysState_Q)$ and substitution σ: If $(P_\sigma, SysState_P) \xrightarrow{a} (P', SysState_P')$ $(bn(a) \notin fn(P_\sigma, Q_\sigma))$, $\exists Q'$ s.t. $(Q_\sigma, SysState_Q) \xrightarrow{a} (Q', SysState_Q')$, $(P', SysState_P')R(Q', SysState_Q')$ and $SysState_P' = SysState_Q'$.

As an independent dimension for system description, we can also exclusively follow the lead of states to define equivalence between systems.

Definition 7 (state simulation). A symmetric binary relation R is a state simulation relation, iff $(P, SysState_P)R(Q, SysState_Q)$ and any substute σ, if $(P_\sigma, SysState_P) \xrightarrow{a} (P', SysState_P')$ $(bn(a) \notin fn(P_\sigma, Q_\sigma))$, $\exists Q'$ s.t. $(Q_\sigma, SysState_Q) \Rightarrow (Q', SysState_Q')$ and $SysState_P' \prec SysState_Q'$.

Definition 8 (state bi-simulation). A symmetric binary relation R is a state bi-simulation relation, iff R and its reverse are both state simulation relations.

3 Formal semantics of grid workflows

3.1 Formalism of services

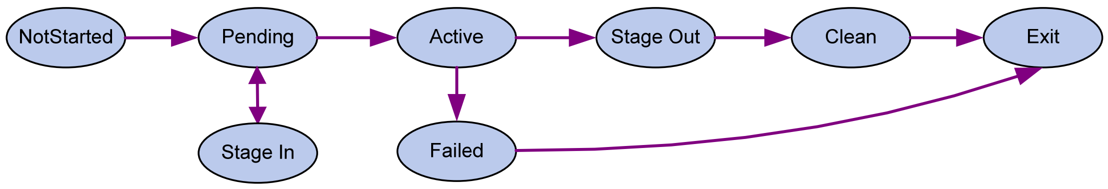


Figure 1 Job State Abstractions for Grid Services

As shown in Figure 1, each service can be *pended* for *staging in* required input data. After the service is executed (i.e. being *active*), it *stages out* the results and *cleans* any unnecessary data, or otherwise the execution of service can *fail*. Therefore, based on state π calculus the service formalism in grid workflows is as follows:

$$
\begin{aligned}
&\#STATE\ \ srvactive = \{A.Status, \{Active\}\};\ \ \#STATE\ \ srvstagingin = \{A.Status, \{StagingIn\}\}\\
&\#STATE\ \ srvpending = \{A.Status, \{Pending\}\};\ \ \#STATE\ \ srvfailed = \{A.Status, \{Failed\}\}\\
&\#STATE\ \ srvexit = \{A.Status, \{Exit\}\};\ \ \#STATE\ \ srvstagingout = \{A.Status, \{StagingOut\}\}\\
&\#STATE\ \ srvcleaning = \{A.Status, \{Cleaning\}\};\ \ \#STATE\ \ execsrv = \{ExecutingSrv, \{A\}\};\\
&ServiceA(port_1, execute, set, \overrightarrow{get_i}, succ, fail, port_2) =_{def}\\
&\quad new\ ack(port_1(v:t_1)\{(+, srvpending), (++, execsrv)\}.(\prod\nolimits_{i=1}^{n} StageIn_i \mid\\
&\quad \underbrace{ack.....ack}_{n-1}.ack\{(+, srvactive)\}.new\ t\ f(\overline{execute} < v:t_1, t, f > \mid\\
&\quad (t(res:t_3).(StageOut \mid ack.\overline{port_2}\{(+, srvexit), (--, execsrv)\} < succ >) +\\
&\quad f\{(+, srvfailed)\}.\overline{port_2}\{(+, srvexit), (--, execsrv)\} < fail >))))\\
&StageIn(get, ack) =_{def}\ \ get(x:t_2)\{(+, srvstagingin)\}.\overline{ack}\{(+, srvpending)\}\\
&StageOut(set, ack, res) =_{def}\\
&\quad new\ res\ clean(\overline{set} < res:t_3 > \{(+, srvstagingout)\}.\overline{clean} \mid clean\{(+, srccleaning)\}.\overline{ack}))
\end{aligned}
$$

In the above formalism for a service *A*, '*#STATE*' is a reserved word for state declarations. According to the syntax of state π calculus, when no state declaration is predefined, states can also be alternatively defined in the declaration of actions. Free names *port*, *set* and *get* are channels for interaction of services and variables (their definition will be given in the next section). Since there are cases in grid systems when concurrent access to expensive resources is not desired, nested process definition is used in the formalism of *ServiceA*. The purpose is to allow the creation of a new instance of process *ServiceA* only when the old instance of *ServiceA* is finished. When multiple instance of a service is desired, the nested position of process *ServiceA* should be changed as follows:

$$
\begin{aligned}
&ServiceA(port, execute, set, get, succ, fail) =\\
&\quad new\ ack(port(v)\{(+, srvpending), (++, execsrv)\}.(...... \mid ServiceA)
\end{aligned}
$$

3.2 Formalism of activities

Grid workflows adopt four basic activities from BPEL4WS: *Receive*, *Send*, *Invoke* and *Assign*. In BPEL4WS, data interactions between activities can be realized by sharing of variables. Consequently, activities of *Receive* and *Send* can be used to model data passing in a grid workflow and the activity of *Assign* can be used to model specific data reproduction. Their formal definitions are provided as follows:

$$Invoke(start, get, port_1, port_2, done) =_{def} \ start.get(v:t).\overline{port_1}<v:t>.port_2(s).\overline{done}$$
$$Receive(start, port_2, set, done) =_{def}$$
$$start\{++,\{msgPort,\{port_2\}\}\}.port_2(v:t)\{--,\{msgPort,\{port_2\}\}\}.\overline{set}<v:t>.\overline{done}$$
$$Reply(start, get, port_1, done) =_{def} \ start.get(v:t).\overline{port_1}<v:t>.\overline{done}$$
$$Assign(start, get_1, set_2, done) =_{def} \ start.get_1(v:t).\overline{set_2}<v:t>.\overline{done}$$
$$Empty(start, done) =_{def} \ start.\overline{done}$$
$$Var_V =_{def} \ Var_{V0}(set, get)$$
$$Var_{V0}(set, get) =_{def} \ set(x_1:t)\{++,\{V.bSize,\{x_1\}\}\}.Var_{V1}(set, get, x_1)$$
$$Var_{V1}(set, get, x_1) =_{def} \ set(x_2:t_2)\{++,\{V.bSize,\{x_2\}\}\}.Var_{V2}(set, get, x_1, x_2) \ +$$
$$\overline{get}<x_1:t_1>\{--,\{V.bSize,\{x_1\}\}\}.Var_{V0}(set, get)$$
......
$$Var_{Vn}(set, get, x_1, ..., x_n) =_{def} \ \overline{get}<x_n:t_n>\{--,\{V.bSize,\{x_n\}\}\}.Var_{Vn-1}(set, get, x_1, ..., x_{n-1})$$

Here the value access and assignment in *Variable*s are realized by channels *get* and *set*. The *Variable* process implements a variable stack with arbitrary depth. In a real grid workflow, the definition of *Variable* can also be simplified as follows if its depth is 1.

$$Var_V(set, get, x) =_{def} \ \overline{get}<x:t>.Var_V(set, get, x) +$$
$$set(y:t)\{++,\{V.CurrentVal,\{y\}\}\}.Var_V(set, get, y)$$

Moreover, all the above activities use the channel of *port* to trigger the execution of a desired service and obtain its result. Note that in BPEL4WS, 'link name', 'partner name' and 'operation name' are three elements in its activities to define the access of a service. A service in grid workflows can thus be first defined as an abstract one and later refined to an executable one by using a service mapping/selection mechanism, as described in [33] and [34]. Therefore the *port* channel here is used to indicate both an abstract service interface (e.g. an abstract functional definition of the service), and a concrete service invocation interface (e.g. via WS-Addressing). The service mapping / selection in grid workflows is further discussed in the next section.

3.3 Service selection

There are often scenarios when multiple candidate services are available to implement a desired abstract function. Semantics of service selection need to be formally defined. A simple way to define interaction with one of candidate services is direct composition of their corresponding state π calculus processes. For the *invocation* of 1-out-of-*n* services, the implementation is as follows:

$$ClosedInvocation = Invoke \mid Service_1 \mid ...... \mid Service_n$$

The above processes of *Invoke, Service*$_1$, ..., *Service*$_n$ share the same *port* channel. In this way multiple services compete for a single *Invoke* activity. The competition is resolved by a non-deterministic choice from *n* services. However, when a specific service selection strategy needs to be explicitly modeled, an addition process for service selection should be implemented:

$$Selection_1(port_{sel}, port_{11}, port_{12}, ..., port_{n1}, port_{n2}) =_{def} \overline{port_{sel}} < port_{11}, port_{12} > .Selection_2$$
......
$$Selection_n(port_{sel}, port_{11}, port_{12}, ..., port_{n1}, port_{n2}) =_{def} \overline{port_{sel}} < port_{n1}, port_{n2} > .Selection_1$$
$Selection =_{def} Selection_1$ n is a predefined constant
$$Invoke'(start, get, port_{sel}, done, fail, succ) =_{def}$$
$$start.get(v:t).port_{sel}(p_1, p_2).\overline{p_1} < v:t > .p_2(s).\overline{done}$$
$$ClosedInvocation =_{def} Invoke' \mid Service_1 \mid ...... \mid Service_n \mid Selection$$

The process of *Selection* stores all *port* channels for the desired abstract function. It selects these *port*s sequentially by their orders in a queue. The order of the *port*s, on the other hand, can be decided by the performance of different corresponding services such as QoS, execution time, etc. Moreover, the new invocation process *Invoke'* no longer interacts directly to a specific service by the given *port*. It queries the *Selection* process first to get what exact service it should invoke by the naming passing capability of π calculus. The interaction between *Invoke'* and the target service can thus be dynamically formed.

3.4 Formalism of workflows

Grid workflows adopt six BPEL4WS control structures: *Sequence*, *While*, *Flow*, *Switch*, *Pick* and *Link*. The formalism of these structures is as follows.

The *Sequence* structure defines sequential relations among execution in a grid workflow:

$$Sequence(fn_d(Act_1), fn_s(Act_2))$$
$$=_{def} Act_1 \, ; Act_2 =_{def} (new\ start_{Act_2})(\{start_{Act_2} / done_{Act_1}\} Act_1 \mid Act_2)$$

The *While* structure defines repeat invoking of one or a group of services in a grid workflow under certain conditions:

$$While(fn_{sd}(Act), start_{while}, done_{while}) =_{def} new\ start_{Act}\ done_{Act}(start_{while}.$$
$$([eval(C,\{t\})](\overline{start_{Act}} \mid Act \mid done_{Act}.(\overline{start_{while}} \mid While)) + [eval(C,\{f\})]\overline{done_{while}}))$$

The *Flow* structure defines synchronization of parallel execution and completion among service activities and structures in a grid workflow:

$$\begin{aligned}
&Flow(fn_{sd}(Act_1), ..., fn_{sd}(Act_m), start_{Flow}, done_{Flow}) =_{def}\\
&\quad (new\ start_{Act_1}\ ...\ start_{Act_m}\ done_{Act_1}\ ...\ done_{Act_m}\ ack\ ack')\\
&\quad (start_{Flow}.Starter \mid Act_1 \mid ...... \mid Act_m \mid Acker \mid ack'.\overline{done_{Flow}})\\
&Starter(start_{Act_1}, ..., start_{Act_m}) =_{def}\ \overline{start_{Act_1}} \mid ...... \mid \overline{start_{Act_m}}\\
&Acker(done_{Act_1}, ..., done_{Act_m}, ack, ack') =_{def}\ done_{Act_1}.\overline{ack} \mid ...... \mid done_{Act_m}.\overline{ack} \mid \underbrace{ack........ack}_{m}.\overline{ack'}
\end{aligned}$$

The *Switch* structure defines a conditional choice structure in a grid workflow:

$$\begin{aligned}
&Switch(fn_{sd}(Act_1), fn_{sd}(Act_2), start_{Switch}, done_{Switch}) =_{def} (new\ start_{Act_1}\ start_{Act_2}\ done_{Act_1}\ done_{Act_2})\\
&\quad (start_{Switch}.([eval(C_1,\{t\})]\overline{start_{Act_1}}\{(++,\{Branch,\{Act_1\}\})\} \mid Act_1 \mid done_{Act_1}.\overline{done_{Switch}} +\\
&\quad [eval(C_1,\{f\}) \wedge eval(C_2,\{t\})]\overline{start_{Act_2}}\{(++,\{Branch,\{Act_2\}\})\} \mid Act_2 \mid done_{Act_2}.\overline{done_{Switch}}))
\end{aligned}$$

The *Pick* structure defines execution selection among different services and structures in a grid workflow based on message trigger:

$$\begin{aligned}
&Pick(fn_{sd}(Act_1), fn_{sd}(Act_2), port_{p1}, port_{p2}, timeout, start_{Pick}, done_{Pick}) =_{def}\\
&\quad (new\ start_{Act_1}\ start_{Act_2}\ done_{Act_1}\ done_{Act_2})(start_{Pick}.\\
&\quad (port_{p1}\{++,\{msgPort_1,\{port_{p1}\}\}\}.\\
&\qquad start_{Act_1}\{(++,\{Event,\{Act_1\}\}),(--,\{msgPort_1,\{port_{p1}\}\})\} \mid Act_1 \mid done_{Act_1}.\overline{done_{Pick}} +\\
&\quad port_{p2}\{++,\{msgPort_2,\{port_{p2}\}\}\}.\\
&\qquad start_{Act_2}\{(++,\{Event,\{Act_2\}\}),(--,\{msgPort_2,\{port_{p2}\}\})\} \mid Act_2 \mid done_{Act_2}.\overline{done_{Pick}}) +\\
&\quad timeout\{(++,\{Event,\{Timeout\}\})\}.\overline{done_{Pick}})
\end{aligned}$$

On the other hand, the *Link* structure imposes synchronization constraints on activities in a grid workflow. Each *Link* has a source and target activity, which restricts that the target activity can only be executed after the source activity is done. Besides, when a ‘death-path’ is detected in a grid workflow (e.g. if a branch in a *Switch* to which the activity *A* belongs is not selected), *negative* tokens should be propagated through all outgoing *Link*s of *A* (i.e. *A* is the source activity of these *Link*s). The semantics are also known as the *Death-Path Elimination* in BPEL4WS. The formalism of *Link* is given in the following:

$$\begin{aligned}
&Link_i(done_{in}, neg_{in}, ack, nack) =_{def}\ EvalTransCondition_i(done_{in}, neg_{in}, ack, nack)\\
&EvalTransCondition_i(done_{in}, neg_{in}, ack, nack) =_{def}\ done_{in}.\overline{ack} + neg_{in}.\overline{nack} \qquad i = 1,...,n\\
&Links(done_{in}, neg_{in}, done_{links}, deathpath) =_{def}\ (new\ ack\ nack)\\
&\quad (Link_1 \mid ... \mid Link_n \mid \underbrace{ack.(...\ .(ack}_{n}.\overline{done_{links}} + nack.\overline{deathpath})... + nack.\overline{deathpath}))\\
&ActWithLinks(freeN) =_{def}\ start.(done_{links}.new\ t\ f(\overline{evaljoin} < t, f > .(\\
&\quad t.(new\ start_{Act}\ done_{Act})(\overline{start_{Act}} \mid Act \mid done_{Act}.\overline{done}.\prod \overline{done_{out}}) +\\
&\quad f\{++,\{Exception,\{Act\}\}\}.\overline{throw} < joinfailure >)) + deathpath.\prod \overline{neg_{out}})\\
&freeN = \{start, done, done_{links}, done_{out}, neg_{out}, deathpath,\\
&\qquad evaljoin, fault, joinfailure, fn_{sd}(Act)\}
\end{aligned}$$

In the above state π calculus process, *ActivityWithLinks* indicates the implementation of the four types of activities introduced in Section 3.2 when *Link* is considered. *ActImpl* is a shortcut notation for detailed formalism of *Receive*, *Send*, *Assign*, *Invoke* activities in Section 3.2. In *Activity-WithLinks*, the start of an activity is subject to completion of its previous activity (*donepreceding*) and incoming *Link*s (*donelinks*). The process then starts to evaluate execution conditions for the corresponding activity (*evalJoin*). The activity will be normally executed if all conditions are satisfied, or otherwise a *JoinFailure* exception is thrown by the *ThrowAct* process (see its implementation in the next section) and the exception is recorded into the *Exception* variable in a grid workflow. Note that in the above *Link* processes, for each received *negativein* token, it will pass the information via the *deathpath* channel such that the *negativeout* token can continue to be propagated to outgoing *Link*s of the corresponding activity.

3.5 Formalism of handling exceptions and compensations

Due to the existence of dynamic interactions and long-running services in grid applications, handling of exceptions and compensations is a critical issue in grid workflows. To correctly depict this aspect of semantics in grid workflows, the *Invoke* activity needs to be further implemented as follows:

$$
\begin{aligned}
&Invoke_S\_WithFault(start, get, port_1, port_2, done, throw, invokefailure, fail, succ) =_{def}\\
&\quad start.get(v:t).\overline{port_1}<v:t>.port_2(s).(\\
&\quad [s = fail]\tau\{(++,\{Exception,\{S\}\})\}.\overline{throw}<invokefailure> + [s = succ]\overline{done})\\
&ThrowAct(throw, fault) =_{def} throw(failttype).\overline{fault}<failttype>
\end{aligned}
$$

When the invocation of a service returns a failure *([u=fail])*, the *ThrowAct* process throws an *invokeFailure* exception and records it into the state *Exception*. On the other hand, the *FaultHandling* process is responsible for capturing and processing the corresponding exceptions. The channel *fault* is used to receive the exception that *ThrowAct* throws out. If the received exception type can be processed by *FaultHandling* (here $type_1$, …, $type_n$ can be the previously mentioned *invokefailure*, *joinfailure*, or other user customized exceptions), corresponding *Activity* is executed to deal with the exception (detailed implementation of *Activity* is omitted here). Otherwise *FaultHandling* sequentially invokes compensation activities to compensate the failure caused by the exception. This is defined in grid workflows as:

$$
\begin{aligned}
&FaultHandling(faulttype, \overline{type}, \overline{compensate}, fn_s(Act_1),\ \ldots, fn_s(Act_n)) =_{def}\\
&\quad fault(faulttype).(new\ start_1\ \ldots\ start_n)\\
&\quad ([faulttype = type_1](\overline{start_1} \mid Act_1) + \ldots + [faulttype = type_n](\overline{start_n} \mid Act_n) +\\
&\quad \sum\nolimits_{type_i \in \overline{type},\ i \neq 1..n} [faulttype = type_i](\overline{compensate_1}.\ \ldots\ldots\ .\overline{compensate_m}))\\
&\quad m \text{ is a finite constant}\\
&CompensationHandler_j(compensate_j, fn_{sd}(Act_j^c)) =_{def}\ compensate_j.\\
&\quad ((new\ start_j^c\ done_j^c)(\overline{start_j^c} \mid Act_j^c \mid done_j^c.CompensationHandler_j)) \qquad 1 \le j \le m
\end{aligned}
$$

3.6 Formalism of global termination

It is required to terminate all service activities that are being (or waiting to be) executed when certain conditions become true (e.g. abnormality in the executing). Different from the Cancellation Patterns proposed by Puhlmann [35], it requires all activities monitor termination signals but rather withdraw the waiting for the service invoke. Meanwhile, another global termination signal is required to ensure proper termination of all activities. Formalism of global termination is described in the following:

$$Invoke(start, get, port_1, port_2, done) =_{def} start.[\phi]get(v:t).[\phi]\overline{port_1} < v:t > .[\phi]port_2(s).[\phi]\overline{done}$$

$$Receive(start, port_2, set, done) =_{def} start.[\phi]port_2(v:t).[\phi]\overline{set} < v:t > .[\phi]\overline{done}$$

$$Reply(start, get, port_1, done) =_{def} start.[\phi]get(v:t).[\phi]\overline{port_1} < v:t > .[\phi]\overline{done}$$

$$Assign(start, get_1, set_2, done) =_{def} start.[\phi]get_1(v:t).[\phi]\overline{set_2} < v:t > .[\phi]\overline{done}$$

The condition ϕ equals to *eval*(*Exception*, {}). Based on the semantics of state π calculus, the process behavior is *null(0)* when the condition is not satisfied. Termination of a process is easily achieved via global management of state π calculus.

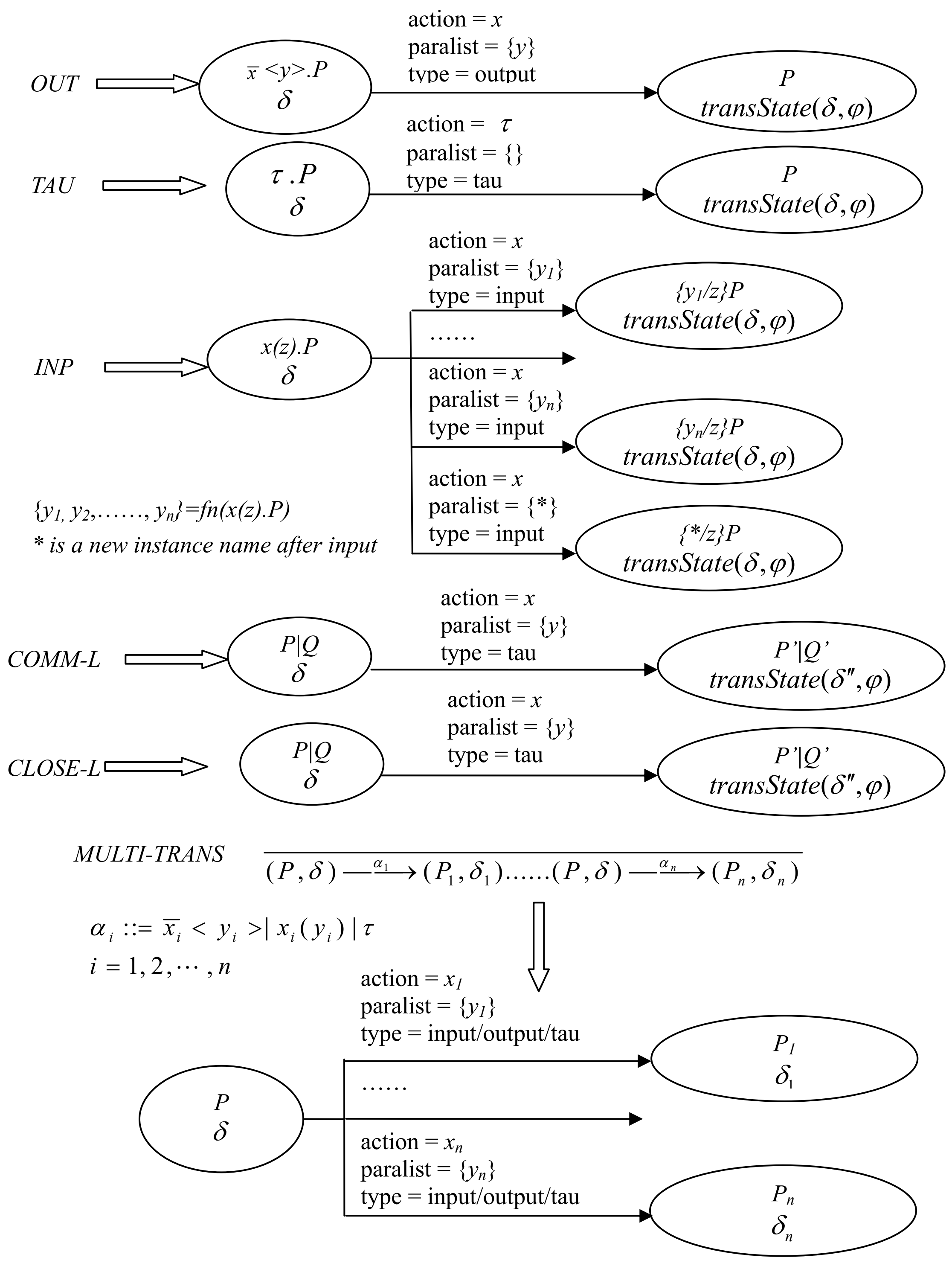
4 Formal verification of grid workflows

4.1 State labeled transition system (SLTS)

Management of actions and behaviors with state π calculus are achieved via the state label transition system. The application of SLTS leads to complete reasoning of grid workflow behaviors in its state space.

The first critical step is the transform from state π calculus formal semantics to the corresponding SLTS. This step is used not only to complete analysis of proposition properties of grid workflows, but also to enable existing model checking techniques incorporated into the framework of state π calculus seamlessly.

In previous sections, it is mentioned that basic π calculus can be interpreted using a general label transition system. In Ferrari [36] and Pistore's [37] work, it is proven that any finite π calculus process can be transformed to its equivalent general label transition system via pre-transition semantics. This is actually a transformation from a name-based to nameless formal theory. For state π calculus, although its operational semantics is also based on pre-transition semantics (see Section 2.3), additional extensions to SLTS have to be processed, e.g. creating and managing state labels when action labels are created as processes evolve. This is the dual label character of SLTS (action labels + state labels). Some critical rules of SLTS are summarized in Figure 2. Operational semantics in Figure 2 are already introduced in Section 2.3.



$$\frac{}{(P,\delta)\xrightarrow{\alpha_1}(P_1,\delta_1)\ldots\ldots(P,\delta)\xrightarrow{\alpha_n}(P_n,\delta_n)}$$

$$\alpha_i ::= \overline{x_i} < y_i > | \, x_i(y_i) \, | \, \tau$$
$$i = 1, 2, \cdots, n$$

SUM-L,PAR-L,RES,OPEN,REP-ACT,REP-COMM,REP-CLOSE can be derived.

Figure 2 SLTS Semantic Transformation Rules

A complete transformation algorithm is illustrated in the flowchart of Figure 3.

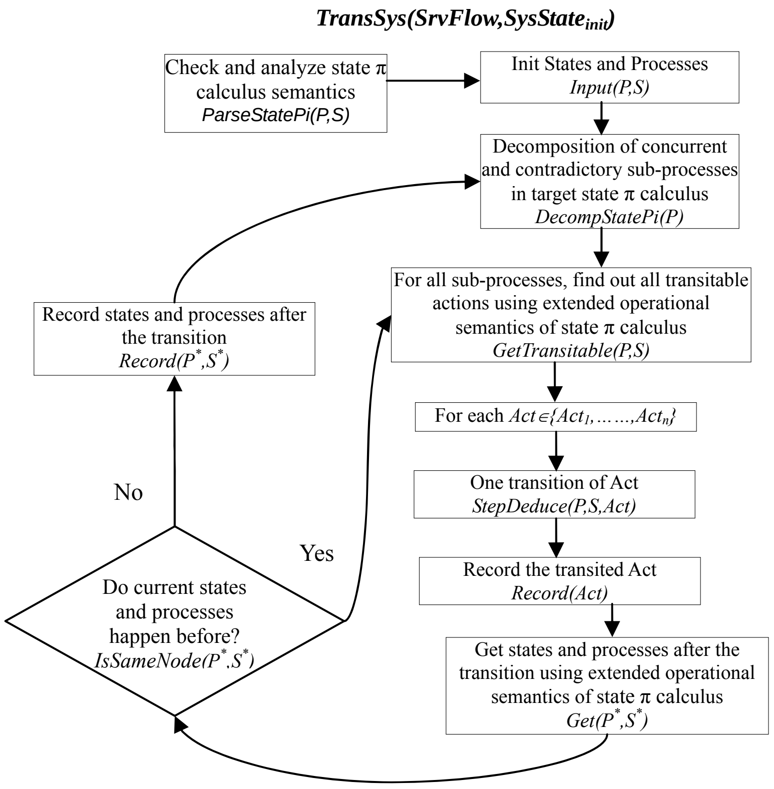

TransSys(SrvFlow,SysState_init)
Check and analyze state π calculus semantics ParseStatePi(P,S)
Init States and Processes Input(P,S)
Decomposition of concurrent and contradictory sub-processes in target state π calculus DecompStatePi(P)
For all sub-processes, find out all transitable actions using extended operational semantics of state π calculus GetTransitable(P,S)
For each Act∈{Act1,……,Actn}
One transition of Act StepDeduce(P,S,Act)
Record the transited Act Record(Act)
Get states and processes after the transition using extended operational semantics of state π calculus Get(P*,S*)
Do current states and processes happen before? IsSameNode(P*.S*)
Yes
No
Record states and processes after the transition Record(P*,S*)

Figure 3 Flowchart for SLTS Transformation

4.2 Structural verification

Structural verification is a fundamental stage in formal verification of grid workflows. Reachability and terminatability are two aspects considered in structural verification. More specifically, given the context of a grid workflow, reachability checks whether there is some service that cannot be arrived due to restraints in the given service set; terminatability checks whether a given termination condition can be met.

Definition 9 (execution). A tuple (α,β) is defined as an execution of state π calculus (P,S), if:

- α is an ordered finite state π calculus action sequence $\alpha=\{\pi_1\{StateExpr_1\}, \pi_2\{StateExpr_2\},\ldots,\pi_n\{StateExpr_n\}\}$;
- β is a finite state sequence $\beta=\{S_1, S_2,\ldots, S_n\}$ corresponding to α;
- $(P,S)\xrightarrow{\pi_1\{StateExpr_1\}}(P_1,S_1)\xrightarrow{\pi_2\{StateExpr_2\}}(P_2,S_2)\ldots\ldots\xrightarrow{\pi_n\{StateExpr_n\}}(P_n,S_n)$, in which $(P_i, S_i) \neq (P_j, S_j)$, $(1\leq i\leq n,\ 1\leq j\leq n,\ i\neq j)$

Here we call P and S initial processes and states, P_n and S_n end processes and states.

Definition 10 (acceptable execution). A tuple (α,β) is an acceptable execution of state π calculus process (P,S), if:

- (α,β) is an execution of (P,S);
- There is no other execution (α',β'), where $\alpha\subset\alpha'$ and $\beta\subset\beta'$.

So here in state π calculus, an accepted execution of (P,S) is the longest transition process without looping in its corresponding SLTS.

Definition 11 (strong state assertation). For $(P,S) \models \lceil Sc \rfloor_{(P',S')}$, $\forall(\alpha,\beta)$, which are acceptable executions of (P,S), $\forall(P,S)\xrightarrow{\alpha^*}(P^*,S^*)$, where $\alpha^*\subset\alpha$, if $P^*\equiv P'$ and $S^*\rightarrow S'$, there is $S^*\rightarrow Sc$. In this situation, the state π calculus process (P,S) satisfies a strong state assertation Sc defined on the targeting process (P',S'), $(P,S)\models\lceil Sc\rfloor_{(P',S')}$.

Definition 12 (weak state assertation). $\exists(\alpha,\beta)$, which are acceptable executions of (P,S), $\exists(P,S)\xrightarrow{\alpha^*}(P^*,S^*)$, where $\alpha^*\subset\alpha$, if $P^*\equiv P'$ and $S^*\rightarrow S'$, there is $S^*\rightarrow Sc$. In this situation, the state π calculus process (P,S) satisfies a weak state assertation Sc defined on the targeting process (P',S'), $(P,S)\models\langle Sc\rangle_{(P',S')}$.

Definition 13 (reachability). Given a state π calculus of a grid workflow *SrvFlow* and its initial state $SysState_{init}$, a set of service $SRV=\{Srv_1,\ldots, Srv_n\}$ is reachable, if $\forall Srv\in SRV$, $(SrvFlow, SysState_{init})\models\langle Srv.Status=Exit\rangle_{(\Phi,\, true)}$.

Definition 14 (terminability). Given a state π calculus of a grid workflow *SrvFlow* and its initial state $SysState_{init}$, *SrvFlow* is terminatable under termination conditions *TC*, if $(SrvFlow, SysState_{init})\models\lceil Tc\rfloor_{(\Phi,\, true)}$.

4.3 Semantic restraint verification

Semantic restraint verification of grid workflows are used to ensure that the model we use is not contradictory to related restraints. Some of these restraints can be checked by its intuitive syntax (for example, in BPEL4WS *Link* structure can't form into a loop). In this work, two types of semantic restraints that cannot be directly verified from its syntax are focused, message competitive confliction and variable garbage collection.

More specifically, it's explicitly announced in BPEL4WS that any service instance shouldn't trigger two or more receiving activities to monitor one event sent from a same port to avoid message conflictions. Variable garbage collection means in the execution process of grid workflows, all temporary variables should be null at the end to ensure no extra message and data.

Definition 15 (message competitive confliction). Given a state π calculus of a grid workflow *SrvFlow* and its initial state $SysState_{init}$, no message competitive confliction exists during its execution, if $\forall msgPort$ during transitions of *SrvFlow*, $(SrvFlow,SysState_{init})\models\lceil |range(*.msgPort)|=0 \vee |range(*.msgPort)|=1\rfloor_{(—,\, true)}$.

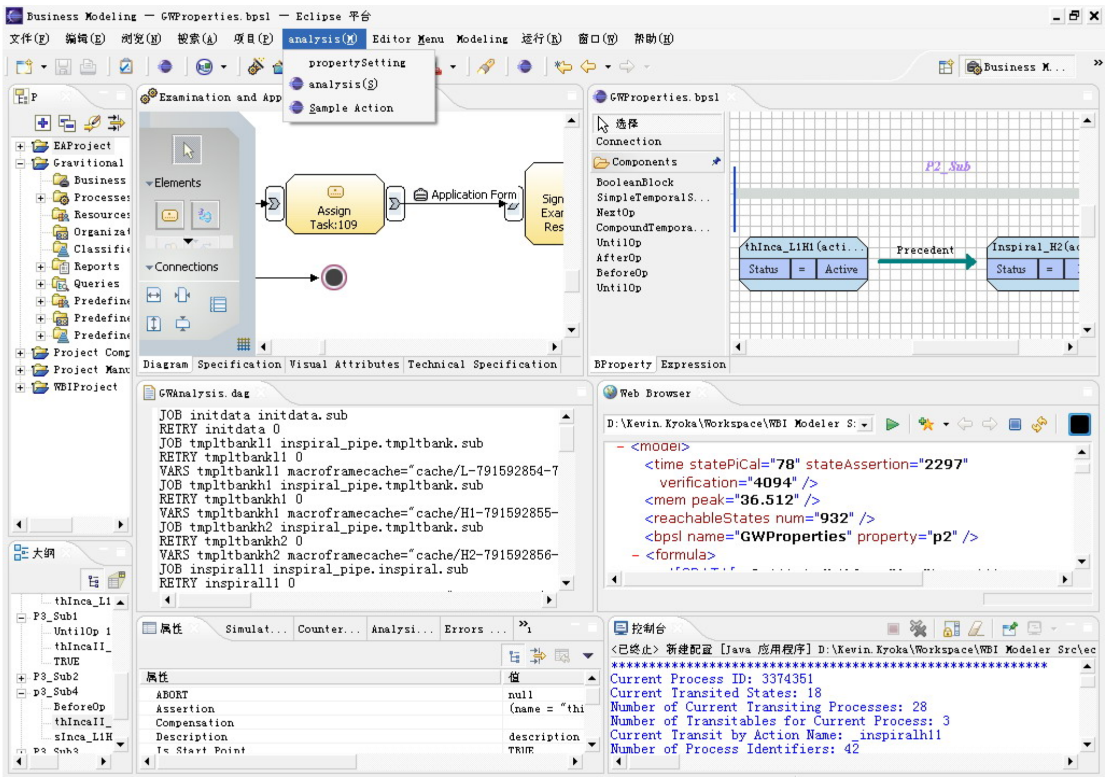
Definition 16 (variable garbage collection). Given a state π calculus of a grid workflow *SrvFlow* and its initial state $SysState_{init}$, no variable garbage exists during its execution, if $\forall bSize$ during transitions of *SrvFlow*, $(SrvFlow,SysState_{init})\models \lceil |range(*.bSize)|=0\rfloor_{(\Phi,\, true)}$.

5 GridPiAnalyzer

As discussed in previous sections, the correctness and reliability assurance is a critical task for QoS supports of grid workflows. More specifically, the correctness of a grid workflow refers to that it must satisfy all the desired properties and constraints from users; the reliability of a grid workflow refers to that it will loyally fulfill users' requirements without any exceptions during the execution. Based on the formal method for grid workflow QoS proposed in this work, a system implementation is introduced in this section, followed by a detailed case study.

5.1 System implementation

Briefly speaking, model checking consists of three steps: system modeling, property specification and property verification. State π calculus is used as a formal language for modeling grid workflows in this work, and the Linear Temporal Logic (LTL) is used as the property specification language. An automatic verification prototype, namely the GridPiAnalyzer, for grid workflows models is implemented. State π calculus semantics of grid workflows are transformed in GridPiAnalyzer and verification is actually carried out using a mainstream open source engine NuSMV2 [38]. Final results are also additionally encapsulated in GridPiAnalyzer. User interfaces based on the Eclipse platform are illustrated in Figure 4.



GWAnalysis.dag
```
JOB initdata initdata.sub
RETRY initdata 0
JOB tmpltbankl1 inspiral_pipe.tmpltbank.sub
RETRY tmpltbankl1 0
VARS tmpltbankl1 macroframecache="cache/L-791592854-7
JOB tmpltbankh1 inspiral_pipe.tmpltbank.sub
RETRY tmpltbankh1 0
VARS tmpltbankh1 macroframecache="cache/H1-791592855-
JOB tmpltbankh2 inspiral_pipe.tmpltbank.sub
RETRY tmpltbankh2 0
VARS tmpltbankh2 macroframecache="cache/H2-791592856-
JOB inspirall1 inspiral_pipe.inspiral.sub
RETRY inspirall1 0
```

属性	值
ABORT	null
Assertion	(name = "thi
Compensation	
Description	description
Is Start Point	TRUE

```
D:\Kevin.Kyoka\Workspace\WBI Modeler S:
- <model>
    <time statePiCal="78" stateAssertion="2297"
      verification="4094" />
    <mem peak="36.512" />
    <reachableStates num="932" />
    <bpsl name="GWProperties" property="p2" />
  - <formula>
```

```
<已终止> 新建配置 [Java 应用程序] D:\Kevin.Kyoka\Workspace\WBI Modeler Src\ec
************************************************************
Current Process ID: 3374351
Current Transited States: 18
Number of Current Transiting Processes: 28
Number of Transitables for Current Process: 3
Current Transit by Action Name: _inspiralh11
Number of Process Identifiers: 42
```

Figure 4 GridPiAnalyzer User Interfaces

JavaCC is used in GridPiAnalyzer to check the syntax and model compiling. Correspondently, when it finished compiling, GridPiAnalyzer caches grid workflows state π calculus semantics in the meta-model included in Figure 5.

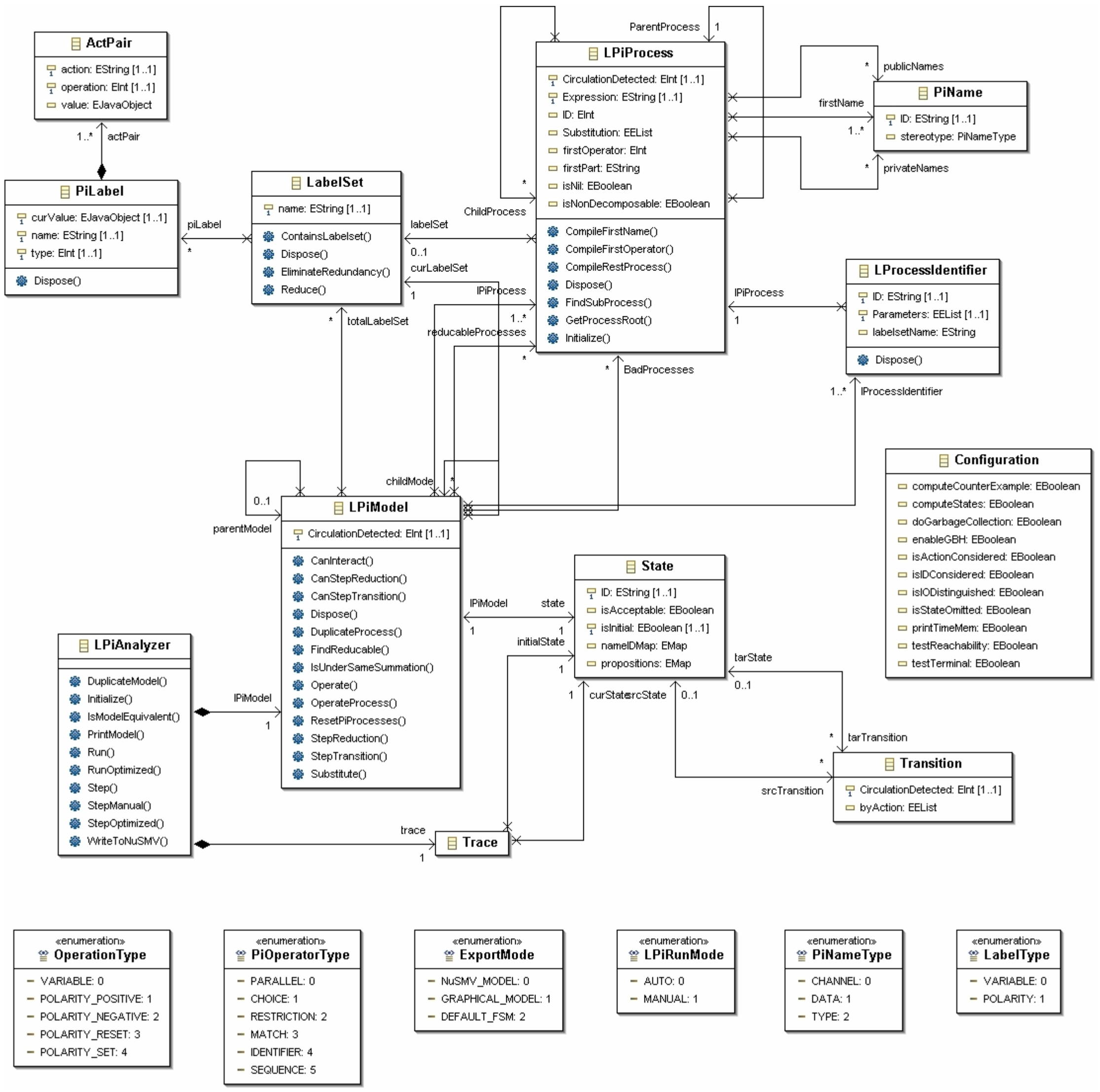


Figure 5 The Meta-Model of State π Calculus Syntax

SLTS transferring, state ascertaining and formal verification are then carried out. Different output results are encapsulated in XML files. It includes criteria of grid workflow models to be tested, process logics to be tested, final results and counter examples.

5.2 A case study – gravitational wave data analysis

5.2.1 Application background

Gravitational Waves (GW) are produced by the movement of energy in mass of dense material which fluctuate space-time structure. The analysis of unknown mass movement and formulation in the universe is stemmed from its detection. But the difficulty is that the detection and analysis of them relates to multiple tasks and massive data.

LIGO (Laser Interferometer Gravitational-Wave Observatory) includes three most sensitive GW detectors in the world, jointly built by Caltech and MIT. LIGO

Scientific Collaboration (LSC) includes over 500 research scientists from over 50 institutes all over the world who are working hard on LIGO data analysis for GW detection. LIGO produces one terabyte of data per day and LIGO data analysis require large amount of CPU cycles. The LIGO data grid [6] provides such a computing infrastructure to integrate petabytes of data storage capability and thousands of CPUs and enable research collaboration cross multiple institutes.

A typical example of a grid workflow for LIGO data analysis can be found in [26]. Figure 6 includes a Condor DAGman script for inspiral GW search and its visualization.

```
JOB initdata initdata.sub
RETRY initdata 0
JOB tmpltbankl1 inspiral_pipe.tmpltbank.sub
RETRY tmpltbankl1 0
VARS tmpltbankl1 macroframecache="cache/L-791592854-791607098.cache" macrochannelname="L1:LSC-AS_Q" macrocalibrationcache="cache_files/
calibration.cache"
JOB tmpltbankh1 inspiral_pipe.tmpltbank.sub
RETRY tmpltbankh1 0
VARS tmpltbankh1 macroframecache="cache/H1-791592855-791607099.cache" macrochannelname="H1:LSC-AS_Q" macrocalibrationcache="cache_files/
calibration.cache"
JOB tmpltbankh2 inspiral_pipe.tmpltbank.sub
RETRY tmpltbankh2 0
VARS tmpltbankh2 macroframecache="cache/H2-791592856-791607100.cache" macrochannelname="H2:LSC-AS_Q" macrocalibrationcache="cache_files/
calibration.cache"
JOB inspirall1 inspiral_pipe.inspiral.sub
RETRY inspirall1 0
VARS inspirall1 macrocalibrationcache="cache_files/calibration.cache" macrobankfile="L1-TMPLTBANK-791592862-2048.xml" macrochannelname="L1:LSC-
AS_Q" macrochisqthreshold="20.0" macroframecache="cache/L-791592854-791607098.cache" macrosnrthreshold="7.0"
JOB trigbankh11 inspiral_pipe.trig.sub
RETRY trigbankh11 0
VARS trigbankh11 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H1:LSC-AS_Q"
JOB trigbankh12 inspiral_pipe.trig.sub
RETRY trigbankh12 0
VARS trigbankh12 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H1:LSC-AS_Q"
JOB inspiralh11 inspiral_pipe.inspiral.sub
RETRY inspiralh11 0
VARS inspiralh11 macrocalibrationcache="cache_files/calibration.cache" macrobankfile="H1-TMPLTBANK-791592863-2049.xml" macrochannelname="H1:LSC-
AS_Q" macrochisqthreshold="20.0" macroframecache="cache/FData-H1-791592855-791607099.cache" macrosnrthreshold="7.0"
JOB inspiralh12 inspiral_pipe.inspiral.sub
RETRY inspiralh12 0
VARS inspiralh12 macrocalibrationcache="cache_files/calibration.cache" macrobankfile="H1-TMPLTBANK-791592864-2050.xml" macrochannelname="H1:LSC-
AS_Q" macrochisqthreshold="20.0" macroframecache="cache/FData-H1-791592855-791607099.cache" macrosnrthreshold="7.0"
JOB sincalih1 inspiral_pipe.sinca.sub
RETRY sincalih1 0
VARS sincalih1 macroframecache="cache/L-791592854-791607098.cache, cache/H1-791592855-791607099.cache"
JOB thincalih1 inspiral_pipe.thinca.sub
RETRY thincalih1 0
VARS thincalih1 macroframecache="cache/L-791592854-791607098.cache, cache/H1-791592855-791607099.cache"
JOB trigbankh21 inspiral_pipe.trig.sub
RETRY trigbankh21 0
VARS trigbankh21 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H2:LSC-AS_Q"
JOB trigbankh22 inspiral_pipe.trig.sub
RETRY trigbankh22 0
VARS trigbankh22 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H2:LSC-AS_Q"
JOB trigbankh23 inspiral_pipe.trig.sub
RETRY trigbankh23 0
VARS trigbankh23 macrocalibrationcache="cache_files/calibration.cache" macrochannelname="H2:LSC-AS_Q"
JOB InspVeto inspiral_pipe.veto.sub
RETRY InspVeto 0
VARS InspVeto macrocalibrationcache="cache_files/calibration.cache" macrochannelname="L1:LSC-AS_Q"
JOB inspiralh21 inspiral_pipe.inspiral.sub
RETRY inspiralh21 0
VARS inspiralh21 macrocalibrationcache="cache_files/calibration.cache" macrobankfile="H2-TMPLTBANK-791592865-2051.xml" macrochannelname="H2:LSC-
AS_Q" macrochisqthreshold="20.0" macroframecache="cache/FData-H2-791592856-791607100.cache" macrosnrthreshold="7.0"
JOB inspiralh22 inspiral_pipe.inspiral.sub
RETRY inspiralh22 0
VARS inspiralh22 smacrocalibrationcache="cache_files/calibration.cache" macrobankfile="H2-TMPLTBANK-791592866-2052.xml"
macrochannelname="H2:LSC-AS_Q" macrochisqthreshold="20.0" macroframecache="cache/FData-H2-791592856-791607100.cache" macrosnrthreshold="7.0"
JOB thinca2lih1 inspiral_pipe.thinca2.sub
RETRY thinca2lih1 0
VARS thinca2lih1 macroframecache="cache/L-791592854-791607098.cache, cache/H1-791592855-791607099.cache"
JOB thinca2lih2 inspiral_pipe.thinca2.sub
RETRY thinca2lih2 0
VARS thinca2lih2 macroframecache="cache/L-791592857-791607101.cache, cache/H1-791592855-791607099.cache"
JOB returnres returnres.sub
RETRY returnres 0

PARENT initdata CHILD tmpltbankl1 tmpltbankh1 tmpltbankh2
PARENT tmpltbankl1 tmpltbankh1 tmpltbankh2 CHILD inspirall1
PARENT inspirall1 CHILD trigbankh11 trigbankh12 thincalih1
PARENT trigbankh11 CHILD inspiralh11
PARENT trigbankh12 CHILD inspiralh12
PARENT inspirall1 inspiralh11 inspiralh12 CHILD sincalih1
PARENT sincalih1 CHILD thincalih1 trigbankh21
PARENT thincalih1 CHILD trigbankh22 returnres
PARENT trigbankh21 CHILD inspiralh21
PARENT trigbankh22 CHILD inspiralh22
PARENT inspiralh21 inspiralh22 CHILD thinca2lih1
PARENT thinca2lih1 thincalih1 CHILD returnres
```

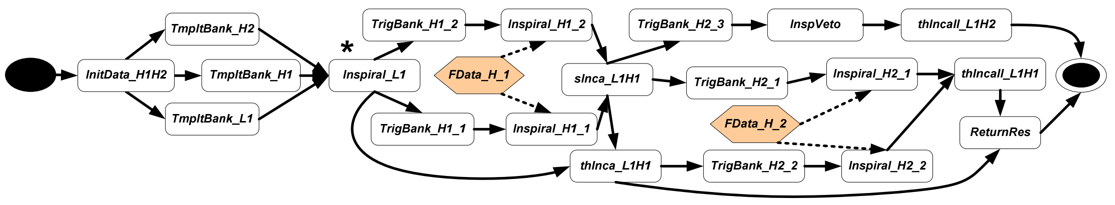


Figure 6 An Example Grid Workflow for GW Search

5.2.2 Grid workflow modeling

In this section, an example of state π calculus semantics for modeling GW data analysis workflows is provided in Figure 7. It is a simplified segment of the workflow described in Figure 6.

JOB *Inspiral inspiral_pipe.inspiral.sub* VARS *VarInsp* ⋯⋯ RETRY *Inspiral* 0
JOB *sInca inspiral_pipe.sinca.sub* VARS *VarsInca* ⋯⋯ RETRY *sInca* 0
JOB *thInca inspiral_pipe.thinca.sub* VARS *VarthInca* ⋯⋯ RETRY *thVar* 0
PARENT *Inspiral* CHILD *sInca thInca*

$$\# STATE\ \ InitS = \{(Inspiral.status, \{NotStarted\}),$$
$$(sInca.status, \{NotStarted\}), (thInca.status, \{NotStarted\})\};$$
$$Invoke_{Inspiral}(start_{Insp}, get_{Insp}, port_{Insp1}, port_{Insp2}, done_{Insp}) =_{def}$$
$$start_{Insp}.get_{Insp}(v).\overline{port_{Insp1}} < v > .port_{Insp2}.\overline{done_{Insp}}$$
$$Invoke_{sInca}(start_s, get_s, port_{s1}, port_{s2}, done_s) =_{def}$$
$$start_s.get_s(v).\overline{port_{s1}} < v > .port_{s2}.\overline{done_s}$$
$$Invoke_{thInca}(start_{th}, get_{th}, port_{th1}, port_{th2}, done_{th}) =_{def}$$
$$start_{th}.get_{th}(v).\overline{port_{th1}} < v > .port_{th2}.\overline{done_{th}}$$
$$Var_{Insp}(set_{Insp}, get_{Insp}, inspd) =_{def}\ \ \overline{get_{GWD}} < inspd > .Var_{Insp}(set_{Insp}, get_{Insp}, inspd)$$
$$+ set_{Insp}(y)\{+, \{Insp.CurrentVal, \{y\}\}\}.Var_{Insp}(set_{Insp}, get_{Insp}, y)$$
$$Var_{sInca}(set_s, get_s, sIncad) =_{def}\ \ \overline{get_s} < sIncad > .Var_{sInca}(set_s, get_s, sIncad)$$
$$+ set_s(y)\{+, \{SInca.CurrentVal, \{y\}\}\}.Var_{sInca}(set_s, get_s, y)$$
$$Var_{thInca}(set_{th}, get_{th}, thIncad) =_{def}\ \ \overline{get_{th}} < thIncad > .Var_{thInca}(set_{th}, get_{th}, thIncad)$$
$$+ set_{th}(y)\{+, \{ThInca.CurrentVal, \{y\}\}\}.Var_{thInca}(set_{th}, get_{th}, y)$$
$$SynPar(freeN) =_{def}\ \ (new\ done_{Insp}\ \ start_s\ \ start_{th})$$
$$(Invoke_{Inspiral} \mid Invoke_{sInca} \mid Invoke_{thInca} \mid SynParImpl)$$
$$freeN = \{start_{Insp}, get_{Insp}, port_{Insp1}, port_{Insp2}, get_s, port_{s1}, port_{s2},$$
$$get_{th}, port_{th1}, port_{th2}, done_s, done_{th}\}$$
$$SynParImpl(done_{Insp}, start_s, start_{th}) =_{def}\ \ done_{Insp}.(\overline{start_s} \mid \overline{start_{th}})$$
$$ServiceFlow =_{def}\ \ new\ \ (fn(SynPar, Var_{Insp}, Var_{sInca}, Var_{thInca}, Service_{Insp}, Service_{sInca}, Service_{thInca}))$$
$$(\tau\{+, InitS\}.(\overline{start_{Insp}} \mid SynPar \mid Var_{Insp} \mid Var_{sInca} \mid Var_{thInca} \mid$$
$$Service_{Insp} \mid Service_{sInca} \mid Service_{thInca} \mid done_s \mid done_{th}))$$

Figure 7 An Example State π Calculus Semantics for GW Data Analysis Workflows

5.2.3 Logic verification for grid workflows

The analysis of GW data involves multiple tasks and large amount of data. Because of the large scale scripts produced by LIGO data analysis tasks which in general may run from days to months, ensuring correctness and effectiveness of workflow structures and logics become critical which can be implemented using GridPiAnalyzer.

The GW detection is a complex task. To distinguish potential GW signals from noises, the whole process can be categorized into four critical logics. Model checking in state π calculus is used to verify these logics. As previously mentioned, model checking is a formal verification tool to analyze expected sequential logics be true or not in finite state system model. Structural verification and formal semantics restraints description of the four groups of LTL logics are listed below.

Logic 1 (Operations after creation of template banks): In any circumstances, once a template bank (*TmpltBank*) is created, two critical following steps: Matching with expected waves (*Inpiral*) and optimizing the matching (*TrigBank*) should be conducted to ensure effectiveness of data analysis.

1) $\mathrm{G}\,(TmpltBank\_H1.Exit \rightarrow ((\mathrm{F}\ TrigBank\_H1.Exit) \wedge (\mathrm{F}\ Inspiral\_H1.Exit)))$
2) $\mathrm{G}\,(TmpltBank\_H2.Exit \rightarrow ((\mathrm{F}\ TrigBank\_H2.Exit) \wedge (\mathrm{F}\ Inspiral\_H2.Exit)))$

Logic 2 (Working state restraints of interferometers): Because of different sensitivity of different interferometers, two interferometers in Handford (H1 and H2) are working simultaneously (*InitData_H1H2*), it is required that the process of matching with the expected wave of H2 (*Inspiral_H2*) should be suspended, until both the data in H1 and H2 pass the contingency analysis (*sInca_L1H1* and *thInca_L1H1*).

$$\mathrm{G}\,((InitData\_H1H2.Active) \rightarrow ((\neg\ Inspiral\_H2.Exit\ \mathrm{U}\ sInca\_L1H1.Active) \wedge (\neg\ Inspiral\_H2.Exit\ \mathrm{U}\ thInca\_L1H1.Active)))$$

Logic 3 (Integrity of contingency analysis): The data collected by three interferometers have to pass all contingency analysis (*sInca, thInca and thIncall*) to minimize noise signal in final analysis. What's more, *sInca* and *thInca* should be done before *thIncall.*

$$((\mathrm{F}\ sInca\_L1H1.Active \wedge (\neg thIncaII\_L1H1.Exit\ \mathrm{U}\ sInca\_L1H1.Active)) \vee (\mathrm{F}\ thInca\_L1H1.Active \wedge (\neg thIncaII\_L1H1.Exit\ \mathrm{U}\ thInca\_L1H1.Active))) \wedge \mathrm{F}\ thIncaII\_L1H1.Exit$$

Logic 4 (Inevitability of contingency analysis): In any circumstance, once the process of matching with expected waves is done or template banks are created, contingency analysis should be done finally.

1) $\mathrm{G}\,(Inspiral\_H1.Exit \ \rightarrow\ (\mathrm{F}\ thIncaII\_L1H1.Exit))$
2) $\mathrm{G}\,(Inspiral\_H2.Exit \ \rightarrow\ (\mathrm{F}\ thIncaII\_L1H1.Exit))$
3) $\mathrm{G}\,(TmpltBank\_H1.Exit \ \rightarrow\ (\mathrm{F}\ thIncaII\_L1H1.Exit))$
4) $\mathrm{G}\,(TmpltBank\_H2.Exit \ \rightarrow\ (\mathrm{F}\ thIncaII\_L1H1.Exit))$

Reachability: $(SrvFlow,\ SysState_{init}) \vDash \langle Srv.Status{=}Exit\rangle_{(\Phi,\ true)}$, in which $Srv \in \{$*initData_H1H2*, *tmpltBank_L1*, *tmplt Bank_H1*, *tmplt Bank_H2*, *Inspiral_L1*, *Inspiral_H1*, *Inspiral_H1*, *TrigBank_H1*, *TrigBank_H2*, *sInca_L1H1*, *thInca_L1H1*,

thIncaII_L1H1, *thIncaII_L1H2*, *InspVeto*, *ReturnRes*};
Terminatability: $(SrvFlow, SysState_{init}) \vDash \lceil ReturnRes.Status{=}Exit \rfloor_{(\Phi,\, true)}$;
Message competitive conflicts: no such restraints in this case study;
Variable garbage collection: $(SrvFlow, SysState_{init}) \vDash \lceil |range(*.bSize)|{=}0 \rfloor_{(\Phi,\, true)}$.

In logics described above, *Inspiral_H1.Exit* is the abbreviation for *Inspiral_H1_1.Exit* ∨ *Inspiral_H1_2.Exit*. This also applys to *Inspiral_H2.Exit, TrigBank_H1.Exit and TrigBank_H2.Exit*. After the transition process with SLTS mentioned above in GridPiAnalyzer, the number of reachable states in the final SLTS of above GW search's state π calculus semantics is 932($2^{9.8642}$), the total number of state proposition is 26, including 20 status variables of service activities. The total time and memory usage of the above logic formulas and specific performance is included in Table 1.

Table 1 Performance Evaluation of Logic Verification Using GridPiAnalyzer

	Time (ms)	Memory usage (MB)
Formalism of state π calculus	78	N/A
Calculating reachable states	1750	N/A
Verification of state assertation	2297	N/A
creating anti-cases	1339	N/A
Verification of Logic 1.1	2125	37.237
Verification of Logic 1.2	2688	37.412
Verification of Logic 2	4094	36.512
Verification of Logic 3	3156	37.410
Verification of Logic 4.1	2328	37.352
Verification of Logic 4.2	2500	37.389
Verification of Logic 4.3	2313	36.887
Verification of Logic 4.4	2359	37.837

The final result shows that all services (*InitData, TmpltBank, Inspiral, sInca, thInca, TrigBank, TrigVeto, thIncall, ReturnRes*) in the LIGO GW search workflow are reachable, and under the condition *TC = "ReturnRes.Status = Exit"*, is terminatable. This means the final analysis can be completed without variable garbage. Regarding four groups of designated logic constraints mentioned above, in the LIGO GW search workflow, Logics 1, 3 and 4 can be met, though the verification result shows that there are anti-cases for Logic 2 which includes 51 state transitions. The workflow can then be further improved to avoid these anti-cases and meet requirements of Logic 2. This indicates the motivation of grid workflow verification. Since in general it will take long time and resources to execute these workflows, formal verification could be used to provide information in advance and improve grid workflow QoS.

6 Conclusions

In this work, a new state π calculus is proposed, which facilitates modeling and verifying of grid workflows. Some typical patterns in grid workflows are captured and both static and dynamic formal verification issues are investigated, including structural correctness, specification satisfiability, logic satisfiability and consistency. A grid workflow modeling and verification environment, GridPiAnalyzer, is implemented using formal modeling and verification methods proposed in this work and validated using a grid workflow for gravitational wave data analysis.

As shown in Table 1 of performance evaluation results, time and memory usage of GridPiAnalyzer is still quite high. Ongoing work is focused on performance optimization of grid workflow verification using GridPiAnalyzer. These include development of new formal methods for workflow decomposition based on standard regions. Using regional analysis, complexity of workflow verification could be dramatically decreased due to smaller numbers of states and processes in each relaxed region.

Acknowledgement

This work is sponsored by Ministry of Education of China under the quality engineering project for higher education "National Open Course Integrated Systems", and Ministry of Science and Technology of China under the national 863 high-tech R&D program (grant No. 2006AA10Z237). This work is carried out in collaboration with Prof. Erik Katsavounidis of LIGO (Laser Interferometer Gravitational-wave Observatory) Laboratory at Massachusetts Institute of Technology.

References

[1]. I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, San Francisco, CA USA, 1998.

[2]. I. Foster, C. Kesselman and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", *Int. J. Supercomputer Applications*, Vol. 15, No. 3, pp. 200-222, 2001.

[3]. I. Foster, C. Kesselman, J. M. Nick and S. Tuecke, "Grid Services for Distributed System Integration", *IEEE Computer*, Vol. 35, No. 6, pp. 37-46, 2002.

[4]. J. Cao, S. A. Jarvis, S. Saini and G. R. Nudd, "GridFlow: Workflow Management for Grid Computing", in *Proc. 3rd IEEE/ACM Int. Symp. on Cluster Computing and the Grid*, Tokyo, Japan, pp. 198-205, 2003.

[5]. J. Yu and R. Buyya, "A Taxonomy of Workflow Management Systems for Grid Computing", *J. Grid Computing*, Vol. 3, No. 3-4, pp. 171-200, 2005.

[6]. E. Deelman, C. Kesselman, G. Mehta, L. Meshkat, L. Pearlman, K. Blackburn, P. Ehrens, A. Lazzarini, R. Williams and S. Koranda, "GriPhyN and LIGO, Building a Virtual Data Grid for Gravitational Wave Scientists", in *Proc. 11th IEEE Int. Symp. on High Performance Distributed Computing*, Edinburgh, Scotland, pp. 225-234, 2002.

[7]. K. Xu, Y. Wang and C. Wu, "Ensuring Secure and Robust Grid Applications - From a Formal Method Point of View", *Advances in Grid and Pervasive Computing*, LNCS Vol. 3947, Springer Verlag, pp. 537-546, 2006.

[8]. I. Foster, "What is the Grid? A Three Point Checklist", *GRIDToday*, July 2002.

[9]. K. Keahey, M. E. Papka, Q. Peng, D. Schissel, G. Abla, T. Araki, J. Burruss, S. Feibush, P. Lane, S. Klasky, T. Leggett, D. McCune, and L. Randerson, "Grids for Experimental Science: the Virtual Control Room", in *Proc. 2nd IEEE Int. Workshop on Challenges of Large Applications in Distributed Environments*, pp. 4-11, 2004.

[10]. L. Pearlman, C. Kesselman, S. Gullapalli, B. F. Spencer, J. Futrelle, K. Ricker, I. Foster, P. Hubbard, and C. Severance, "Distributed Hybrid Earthquake Engineering Experiments: Experiences with a Ground-Shaking Grid Application", in *Proc. 13th IEEE Int. Symp. on High Performance Distributed Computing*, pp. 14-23, 2004.

[11]. D. P. Spooner, J. Cao, S. A. Jarvis, L. He, and G. R. Nudd, "Performance-aware Workflow Management for Grid Computing", *The Computer J.*, Special Focus - Grid Performability, Vol. 48, No. 3, pp. 347-357, 2005.

[12]. X. He, X. Sun, and G. von Laszewski, "QoS Guided Min-Min Heuristic for Grid Task Scheduling", *J. Computer Science & Technology*, Vol. 18, No. 4, pp. 442-451, 2003.

[13]. S. Zhang, N. Gu, and S. Li, "Grid Workflow based on Dynamic Modeling and Scheduling", in *Proc. IEEE Information Technology: Coding and Computing*, Vol. 2, pp. 35-39, 2004.

[14]. I. Brandic, S. Benkner, G. Engelbrecht, and R. Schmidt, "QoS Support for Time-Critical Grid Workflow Applications", in *Proc. 1st IEEE Int. Conf. on e-Science and Grid Computing*, pp. 108-115, 2005.

[15]. K. Czajkowski, A. Dan, J. Rofrano, S. Tuecke, and M. Xu, "Agreement-based Grid Service Management (OGSI-Agreement)", *Global Grid Forum*, GRAAP-WG Author Contribution, 2003.

[16]. K. Keahey, T. Araki, and P. Lane, "Agreement-Based Interactions for Experimental Science", in *Proc. Euro-Par 2004 Parallel Processing*, LNCS Vol. 3149, pp. 399-408, 2004.

[17]. H. Zhang, K. Keahey, and W. Allcock, "Providing Data Transfer with QoS as Agreement-based Service", in *Proc. IEEE Int. Conf. on Services Computing*, pp. 344-353, 2004.

[18]. J. Chen and Y. Yang, "Key Research Issues in Grid Workflow Verification and Validation", in *Proc. 4th ACM Australasian Workshop on Grid Computing and e-Research*, Vol. 54, pp. 97-104, 2006.

[19]. E. M. Clarke, Jr. O. Grumberg and D. A. Peled. *Model Checking*. Cambridge, Mass: MIT Press, pp. 1-231, 1999.

[20]. K. Amin, G. von Laszewski, and M. Hategan et. al., "GridAnt: A Client-Controllable Grid Workflow System", in *Proc. 37th IEEE Annual Hawaii International Conference on System Sciences*, pp. 3293-3301, 2004.

[21]. T. Andrews, F. Curbera, and H. Dholakia et al, "Business Process Execution Language for Web Services", Version 1.1, 2003.

[22]. J. Yu and R. Buyya, "A Novel Architecture for Realizing Grid Workflow using Tuple Spaces", in *Proc. 5th IEEE/ACM Int. Workshop on Grid Computing*, Pittsburgh, USA, pp. 119-128, 2004.

[23]. I. Taylor, M. Shields, I. Wang, and R. Philp, "Distributed P2P Computing within Triana: A Galaxy Visualization Test Case", in *Proc. 17th IEEE Int. Parallel & Distributed Processing Symp.*, Nice, France, 2003.

[24]. W. Bausch, C. Pautasso, and G. Alonso, "Programming for Dependability in a Service-Based Grid", in *Proc. 3rd IEEE/ACM Int. Symp. on Cluster Computing and the Grid*, Tokyo, Japan, pp. 164-171, 2003.

[25]. M. Litzkow, M. Livny, and M. Mutka, "Condor - a Hunter of Idle Workstations", in *Proc. 8th Int. Conf. on Distributed Computing Systems*, pp. 104-111, 1988.

[26]. D. A. Brown, P. R. Brady, A. Dietz, J. Cao, B. Johnson, and J. McNabb, "A Case Study on the Use of Workflow Technologies for Scientific Analysis: Gravitational Wave Data Analysis", in I. J. Taylor, D. Gannon, E. Deelman, and M. S. Shields (Eds.), *Workflows for e-Science: Scientific Workflows for Grids*, Springer Verlag, pp. 39-59, 2007.

[27]. R. Milner, *Communicating and Mobile Systems: The Pi Calculus*, Cambridge University Press, 1999.

[28]. W. M. P. vander Aalst, "Pi Calculus versus Petri Nets: Let us Eat 'humble pie' rather than Further Inflate the 'Pi hype'", *BPTrends*, Vol. 3, No. 5, pp. 1-11, 2005.

[29]. H. Smith, "Business Process Management - the Third Wave: Business Process Modeling Language (BPML) and its Pi-calculus Foundations", *Information and Software Technology*, Vol. 45, pp. 1065-1069, 2003.

[30]. S. Chaki, E. M. Clarke, J. Ouaknine et al, "State / Event-based Software Model Checking", in E. A. Boiten, J. Derrick, G. Smith (Eds.), *Integrated Formal Methods*, LNCS Vol. 2999, Springer Verlag, pp. 128-147, 2004.

[31]. R. D. Nicola and F. Vaandrager, "Three Logics for Branching Bisimulation", *J. ACM*, Vol. 42, No. 2, pp. 458-487, 1995.

[32]. D. Sangiorgi and D. Walker, *The Pi-calculus: a Theory of Mobile Processes*, Cambridge University Press, 2001.

[33]. Z. Németh and V. Sunderam, "Characterizing Grids Attributes, Definitions, and Formalisms", *J. Grid Computing*, Vol. 1, No. 1, pp. 9-23, 2003.

[34]. T. Fahringer, J. Qin, and S. Hainzer, "Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language", in *Proc. IEEE Int. Symp. on Cluster Computing and the Grid*, pp. 676-685, 2005.

[35]. F. Puhlmann and M. Weske, "Using the Pi-calculus for Formalizing Workflow Patterns", *Business Process Management*, LNCS Vol. 3649, pp. 153-168, 2005.

[36]. G. L. Ferrari, S. Gnesi, and U. Montanari et al., "A Model-checking Verification Environment for Mobile Processes", *ACM Transactions on Software Engineering and Methodology*, Vol. 12, No. 4, pp. 440-473, 2003.

[37]. U. Montanari and M. Pistore, "Checking Bisimilarity for Finitary Pi-calculus", *Concurrency Theory*, LNCS Vol. 962, Springer Verlag, pp. 42-56, 1995.

[38]. A. Cimatti, E. Clarke, and E. Giunchiglia et al., "NuSMV2: an Open Source Tool for Symbolic Model Checking", *Computer Aided Verification*, LNCS Vol. 2404, Springer Verlag, pp. 359-364, 2002.